Programmable Systolic Arrays

Richard Paul Hughey

Ph.D. Dissertation

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-91-34
May 1991

Programmable Systolic Arrays

Richard Paul Hughey

B. A., Swarthmore College, 1985
B. S., Swarthmore College, 1985
Sc. M., Brown University, 1987

Thesis
Submitted in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy in the Department of Computer Science
at Brown University.

May, 1991

Vita

RICHARD Hughey was born in Philadelphia, Pennsylvania, on July 31, 1963. He worked toward his high school diploma in Swarthmore, Pennsylvania, Ridgewood, New Jersey, Beverly Hills, California, and Frankfurt, Germany, and it was awarded to him by the Village of Ridgewood in 1981. He received the B. A. degree in mathematics with distinction and the B. S. degree in engineering with distinction from Swarthmore College (Swarthmore, Pennsylvania) in 1985. He received the Sc. M. degree in computer science from Brown University (Providence, Rhode Island and Providence Plantations) in 1987. His research interests include VLSI architecture and design, parallel computing, parallel programming, and the applications of computer science to computational biology. He is a member of the IEEE and the IEEE Computer Society, and was inducted by Phi Beta Kappa, Sigma Xi, and Tau Beta Pi as an undergraduate. He can be reached by electronic mail at rph@cs.brown.edu.

Abstract

SYSTOLIC arrays can solve computationally intensive problems many times faster than traditional computers or supercomputers. Because VLSI systems require many months to design, simulate, fabricate, and test, research has turned to programmable systolic arrays. This thesis introduces a general, programmable systolic array architecture: the Systolic Shared Register (SSR) architecture. The Systolic Shared Register architecture preserves the simple communication of single-purpose systolic arrays while providing a fully programmable systolic co-processor.

To test the SSR architecture, I designed, simulated, and had fabricated the Brown Systolic Array. B-SYS is an 8-bit SSR machine: each 16-element register bank in the linear array is shared between two functional units, providing simple and efficient systolic communication. The 85 000-transistor chip worked on first fabrication and a 10-chip, 470-element prototype array performs sequence comparison over 80 times faster than its Intel 80386 host. A custom-designed board could magnify this to over 600 times faster for a 10-chip co-processor. A 32-chip B-SYS system could process over three billion 8-bit operations every second.

Although the SSR paradigm stands by itself, it is instructive to simultaneously consider programming methods and applications. This thesis describes the principle of software fault detection for the automatic generation of fault-tolerant programs, an efficient alternative to rigid hardware methods. Additionally, this thesis introduces the New Systolic Language; NSL is a programming language for systolic co-processors based on data stream computation. With the aid of NSL, several systolic applications are examined in detail, in particular sequence comparison problems from the Human Genome Project. A comparison of B-SYS to existing parallel machines reveals its unequivocal superiority in its targeted area.

Acknowledgments

THIS thesis, like any other, owes a great debt to a considerable number of people. First, I thank my advisor Daniel Lopresti. He prepared the stage for this work with a single-purpose sequence comparison chip, and his comments and criticisms have been a constant influence on this work. I also thank my committee, Steve Reiss and John Savage, whose careful reading greatly improved this thesis. Ken Zadeck provided important critical advice early in the project.

The discussions of computational biology applications have been greatly enhanced by the efforts of Steve Althschul and Tom White to impart knowledge to an ignorant computer scientist.

While at Brown, I have had the pleasure of knowing many fine people, in particular a large crowd of card players always eager for a break (Mark Boddy, Ken Basye, Glenn Caroll, Eugene Charniak, Tony Davis, Moi Lejter, Paul Markowitz, Gail Mitchell, Maury Neiburg, Gene Santos, Solomon Shimony, and Lynn Stein Melnick). A few people who resisted the influences of Bridge and "Oh Hell" include Cheryl Harkness, Tim Johnson, Scott Meyers, Rob Ravenscroft (who, in addition to being an excellent brewer, proved eager to show that all problems are TEX-reducible), Kate Sanders, and Cathy Schevon.

Several VLSI classes endured assignments on various versions of the Brown Systolic Array simulator. I thank Glenn Caroll, Paul Howard, Gail Mitchell, Marion Nodine, and Elizabeth Schriver for their comments.

My pursuit of an advanced degree was influenced and encouraged by four undergraduate professors: David Bowler who introduced me to electronics and digital logic, Charles Grinstead who introduced me to mathematics and academic research, and Charles Kelemen and Steve Platt who introduced me to computer science.

I would like to extend a special thanks to Todd Karakashian and Karen Ohl, for reasons innumerable.

Finally, I enthusiastically thank my sisters Alice and Barbara, and my parents Ginnie and Joe. Without their support and love throughout my life, this thesis would never have been possible.

Contents

Vita i

Abstract iii

Acknowledgments v

1 Introduction 1

1.1 Systolic Arrays 2

1.2 Applications 2

1.3 Contributions 5

1.3.1 A Class of Programmable Systolic Architectures 5

1.3.2 A Systolic Array Implementation 6

1.3.3 Software Fault Detection for Systolic Arrays 6

1.3.4 Programming Methods for Systolic Arrays 6

1.4 Overview of this Thesis 7

2 Related Work 9

2.1 Systolic Algorithms 9

2.2 Systolic Hardware 17

2.3 Systolic Programming Languages 25

3 Systolic Shared Register Architectures 37

3.1 Overview 37

3.2 Architecture Definition 38

3.2.1 Broadcast Instructions 38

3.2.2 Regular Topology 39

3.2.3 Shared Registers for Communication 39

3.2.4 Stream Programming 41

3.2.5 Generalized SSR Machine 42

3.3 An Evaluation of SSR Communication 44

3.4 Programming Techniques 45

3.4.1 Scratch Registers 46

3.4.2 Phased SSR Programs 46

3.5 Conclusions 48

4 Design of the Brown Systolic Array 49
4.1 Architecture . . . 49
4.1.1 B-SYS Arithmetic Logic Unit . . . 50
4.1.2 Architecture Simulation . . . 54
4.2 Design . . . 55
4.2.1 CMOS VLSI . . . 57
4.2.2 Overview of the B-SYS Chip . . . 59
4.2.3 B-SYS Subsystem Design . . . 62
4.3 Design Verification . . . 70
4.3.1 Logic . . . 70
4.3.2 Timing . . . 70
4.3.3 Power . . . 71
4.4 Conclusions . . . 72

5 Fault Testing and Fault Detection 75
5.1 Overview . . . 75
5.2 Fault Testing . . . 76
5.2.1 Chip Screening . . . 76
5.2.2 The Prototype Board . . . 79
5.2.3 Prototype Performance . . . 81
5.2.4 The Ideal Board . . . 82
5.3 Fault Detection . . . 84
5.3.1 Hardware Fault Detection . . . 84
5.3.2 Software Fault Detection . . . 85
5.3.3 Propagation of Fault Information . . . 89
5.3.4 Analysis . . . 90
5.4 Conclusions . . . 90

6 A Framework for Systolic Programming 93
6.1 Overview . . . 93
6.2 NSL Programs . . . 95
6.2.1 Cell Programs . . . 98
6.2.2 Main Programs . . . 100
6.3 The NSL Implementation . . . 103
6.4 Implementation Enhancements . . . 106
6.5 Conclusions . . . 110

7 B-SYS Applications 113
7.1 Sequence Comparison . . . 113
7.1.1 Simple Sequence Comparison . . . 114
7.1.2 Comparison of One String Against Many . . . 115
7.1.3 Sequence Comparison with Gap Penalties . . . 124
7.1.4 Alternate Transcription . . . 126
7.1.5 Protein Matching . . . 132

7.2 Matrix Multiplication 136
7.3 Conclusions 140

8 Conclusions **143**

A B-SIM User's Guide **147**

B B-SYS User's Guide **149**

C Sequence Comparison on the Connection Machine **151**

Bibliography **155**

List of Tables

1.1 The Genetic Code in RNA form. 3

2.1 Systolic algorithms. 14
2.2 Systolic building blocks. 23
2.3 Systolic systems. 24

4.1 Carry lookahead addition. 51
4.2 B-SYS result block control words. 53
4.3 B-SYS generate and propagate control words. 53
4.4 B-SYS pin names. 61

5.1 B-SYS chip statuses. 78
5.2 B-SYS prototype board key. 80
5.3 B-SYS prototype I/O addresses. 81
5.4 B-SYS sequence comparison timings. 82
5.5 Comparison of single fault detection methods. 90

6.1 Static *Config* class members. 99
6.2 NSL operators and functions. 105
6.3 *SStream* member functions. 109

7.1 Differences modulo 256. 114
7.2 B-SYS cell program lengths for various applications. 140

C.1 CM-2 sequence comparison. 152

List of Figures

2.1 Systolic insertion sort. 10
2.2 Systolic implementation of Horner's method. 11
2.3 Comparison of AAC with AGCA. 13
2.4 P-NAC architecture. 17
2.5 Connection Machine processor architecture. 19
2.6 Warp data path. 20
2.7 Massively Parallel Processor architecture. 21
2.8 SDEF program for Horner's method. 29
2.9 Heart's program for Horner's method. 30
2.10 Occam program for Horner's method. 32
2.11 W2 cell program for Horner's method. 33
2.12 Paris program for Horner's method. 34

3.1 Linear SSR processor array. 40
3.2 Systolic Shared Register networks. 43
3.3 Methods of inter-processor communication. 44
3.4 Phased systolic sort. 47

4.1 Segment of the B-SYS array. 50
4.2 B-SYS arithmetic logic unit. 51
4.3 B-SYS instruction word. 52
4.4 B-SYS phased systolic sort. 54
4.5 B-SIM TEX snapshot. 55
4.6 B-SIM Tango animation. 56
4.7 Example CMOS circuits 57
4.8 CMOS layer key. 58
4.9 Floor-plan of the Brown Systolic Array. 60
4.10 B-SYS pin grid array. 61
4.11 B-SYS clocks and diagnostic signals. 62
4.12 Block diagram of functional unit and register bank. 63
4.13 RAM cell and layout. 64
4.14 Propagate logic block. 65
4.15 *R* logic block and operand latches. 66
4.16 One-bit carry block. 66

4.17 The Manchester carry chain. 67
4.18 Block diagram of the control circuitry. 68
4.19 The flag address decoder. 69
4.20 Maximum clock speeds from Crystal analysis. 71
4.21 B-SYS functional unit and register bank 74

5.1 Pattern generator clock phases. 79
5.2 B-SYS prototype board. 80
5.3 Prototype clocking. 81
5.4 B-SYS* ideal board. 83
5.5 Gross replication of computation. 85
5.6 Replicated computation streams for fault detection. 86
5.7 Skewed replicated computation streams. 87
5.8 Interleaved replicated computation streams. 88

6.1 Two systolic streams.. 95
6.2 Overview of New Systolic Language programming. 96
6.3 NSL cell program for Horner's method. 98
6.4 NSL cell program for sorting. 99
6.5 NSL main program for Horner's method. 100
6.6 NSL main program for sorting. 101
6.7 Uncompressed B-SYS code generated for Horner's method. 102
6.8 B-SYS code generated for Horner's method. 102
6.9 B-SYS code generated for sorting. 103
6.10 NSL prototype data objects. 104
6.11 NSL prototype systolic stream objects. 106
6.12 NSL prototype I/O objects. 107
6.13 NSL cell program for sequence comparison. 108
6.14 B-SYS code generated for sequence comparison. 109

7.1 NSL modulo sequence comparison program. 115
7.2 B-SYS modulo sequence comparison program. 116
7.3 B-SYS prototype macro definitions. 117
7.4 B-SYS one-against-one sequence comparison. 118
7.5 NSL many-against-one sequence comparison. 119
7.6 B-SYS code for many-against-one sequence comparison. 120
7.7 Two sequence comparison algorithms on B-SYS. 121
7.8 Sequence comparison on various machines. 122
7.9 Sequence comparison on various machines (resources). 124
7.10 Dynamic programming matrices for gap sequence comparison. 126
7.11 NSL cell program for gap sequence comparison. 127
7.12 NSL main program for gap sequence comparison. 128
7.13 Network comparison. 129
7.14 Comparison network for phenylalanine—leucine. 130

7.15 Uniform comparison network for phenylalanine—leucine. 131
7.16 Wildcard comparison network for phenylalanine—leucine. 131
7.17 NSL amino acid encoding comparison. 132
7.18 NSL protein alignment cell program. 134
7.19 NSL protein alignment main program. 135
7.20 Systolic transitive closure cell. 137
7.21 Transitive closure cell program. 137
7.22 Transitive closure main program. 138
7.23 Transitive closure I/O functions. 139

A.1 The Brown Systolic Array simulator. 148

C.1 CM sequence comparison driving routine. 153
C.2 CM sequence comparison cell program. 154

Chapter 1

Introduction

THE systolic array is perhaps the most significant architectural development inspired by the revolution in very large scale integration (VLSI) fabrication technology. Systolic co-processors exhibit tremendous performance due to the ease with which processing elements may be replicated hundreds or thousands of times on a single VLSI chip. Since the introduction of the systolic paradigm, systolic co-processors have been proposed to solve a wide variety of problems including speech recognition, sorting and searching, image processing, and various dynamic programming algorithms. Unfortunately, the majority of these proposals assume special- or single-purpose processing elements. Each new VLSI building block requires many months to design, simulate, fabricate, and test. Because of this great effort required to construct new arrays, research has turned to programmable systolic arrays. This thesis presents a general, programmable systolic array architecture: the Systolic Shared Register (SSR) design. The Systolic Shared Register architecture preserves the simple communication of single-purpose systolic arrays while providing the user with a fully programmable systolic co-processor.

Programmable hardware co-processors cannot, of course, exist in a vacuum. Although the SSR paradigm could stand by itself, it is much more instructive to simultaneously consider two other aspects of co-processor design: algorithms and programming languages. Any machine constructed without a clear notion of the types of algorithms it is meant to solve will be poorly designed. Once target algorithms are considered, the problem of programming the system must also be investigated.

The majority of this thesis describes the design, implementation, and use of the Brown Systolic Array (B-SYS), a working VLSI implementation of the SSR architecture. B-SYS was specifically designed for the algorithmic needs of molecular biologists but can efficiently execute many other systolic algorithms. This thesis will discuss the types of algorithms that are well suited to the Brown Systolic Array and present a general framework for programming any systolic co-processor.

The remainder of this introduction will first present some general background information — a brief overview of systolic arrays and their applications. Next, the major contributions of this thesis are summarized, and finally the organization of this work is presented.

1.1 Systolic Arrays

Systolic arrays were originally proposed as a class of parallel architectures by H. T. Kung and C. E. Leiserson in 1978.[89,93] The utility of the systolic principle rests in its formalism for mapping algorithms to large processor arrays. Instead of a labyrinth of individually programmed processing elements, the typical special-purpose systolic array consists of one or two types of cells connected in a regular network, such as a line, ring, square mesh, or hexagonal mesh.

Systolic algorithms are designed with the knowledge that individual processing elements are unable, without great cost, to send information to arbitrary or distant locations. In a planar VLSI process, multiprocessors requiring arbitrary communication will use the vast majority of chip area for routing. The point of a systolic implementation is to put as much processing power as possible in as little space as possible. Thus, in the words of Foster and Kung,[50] a systolic algorithm will have the following properties:

- The algorithm can be implemented by only a few different types of simple cells.
- The algorithm's data and control flow is simple and regular, so that cells can be connected by a network with local and regular interconnections. Long distance and irregular communication is thus minimized.
- The algorithm uses extensive pipelining and multiprocessing. Typically, several data streams move at constant velocity over fixed paths in the network, interacting at cells where they meet. In this way a large number of cells are active at one time so that the computation speed can keep up with the data rate.

As a result of these principles, data is written to and read from the processor array only at array boundaries, while computation takes place throughout the array. For example, data might enter on one side and results might exit on the other. The description of a systolic algorithm, therefore, must specify the operation of the systolic cells in addition to the data movement between the cells. Systolic arrays can efficiently manage large quantities of data without the potential for bus contention or network congestion.

1.2 Applications

Systolic arrays were originally applied to digital signal processing, and now linear arrays for convolution and other types of digital filtering and transformation abound.[50] Often, such systems require only a small number of processing elements (for example, a K-tap finite impulse response (FIR) filter requires K processing elements, not some number proportional to the length of the input stream) so that small systolic pipelines can be formed.

Various types of mesh architecture are used for the 2-dimensional image processing versions of the basic digital signal processing filters and transformations. Many matrix operations can also be performed on mesh systolic arrays.

Combinatorial problems, those requiring only integers and simple operations, present the largest achievable gains with systolic processing. Since such problems only require

UUU	Phenylalanine	UCU	Serine	UAU	Tyrosine	UGU	Cysteine
UUC		UCC		UAC		UGC	
UUA	Leucine	UCA		UAA	Stop	UGA	Stop
UUG		UCG		UAG		UGG	Tryptophan
CUU		CCU	Proline	CAU	Histidine	CGU	Arginine
CUC		CCC		CAC		CGC	
CUA		CCA		CAA	Glutamine	CGA	
CUG		CCG		CAG		CGG	
AUU	Isoleucine	ACU	Threonine	AAU	Asparagine	AGU	Serine
AUC		ACC		AAC		AGC	
AUA		ACA		AAA	Lysine	AGA	Arginine
AUG	Methionine	ACG		AAG		AGG	
GUU	Valine	GCU	Alanine	GAU	Aspartic acid	GGU	Glycine
GUC		GCC		GAC		GGC	
GUA		GCA		GAA	Glutamic acid	GGA	
GUG		GCG		GAG		GGG	

Table 1.1: The Genetic Code in RNA form.

simple operations, small processing elements are sufficient. Only small processing elements can form the largest, most parallel machines. Many combinatorial problems (including some image processing applications) can efficiently use hundreds of thousands or even millions of processing elements, a complexity not achievable with complicated processors or routing schemes.

The strongest influence on the design of the Brown Systolic Array has been the combinatorial problems of the Human Genome Project, biology's first "big science" project.[22,33,34,157,165] The project's goal is to determine the exact structure and function of human DNA (deoxyribonucleic acid).[a] Nucleic acids such as DNA are macromolecules, long sequences of nucleotides, that encode an organism's genetic data. The DNA located in a cell encodes the structure of an organism in thousands or millions of nucleotides — three billion in the case of Homo sapiens. Ribonucleic acid (RNA), which can serve as an intermediate step in the production of proteins from the DNA (in which case it is called messenger RNA), contains up to a few thousand nucleotides. The four nucleotides composing a DNA molecule are: adenine (A), guanine (G), cytosine (C), and thymine (T). Uracil (U) is substituted for thymine in the case of RNA. In computer science terms, a DNA molecule is a string of characters from the four-character alphabet $\Sigma = \{A, G, C, T\}$, while for RNA, $\Sigma = \{A, G, C, U\}$.

A nucleic acid chain may encode many amino acid sequences (proteins). Every three nucleotides, a *codon*, specify which amino acid should be added to the amino acid chain. Each protein is generated from the DNA with a complex chemical decryption algorithm involving the messenger RNA. The codes for each amino acid in terms of the RNA nucleotides are shown in Table 1.1; protein specifications average 1000 codons in length

[a] Although all humans are different, these differences are the result of slight variations in the DNA.

or, equivalently, proteins are chains of around 1000 amino acids.

Nucleic and amino acid sequence data are useless without tools for their analysis. Perhaps the most commonly desired analysis is the determination of the similarity between two pieces of DNA, either to locate where they overlap (so that partial experimental data may be joined) or where they differ (to find, for example, the region responsible for a genetic disorder). Computational biologists have several metrics for comparison which involve different costs between nucleotides as well as the possible use of affine costs or gap penalties to indicate a cost for the deletion of large sections of genetic information. Similar analysis can be performed over the 20-character alphabet of amino acids to determine evolutionary distances between proteins.[15,105,135,154]

Exact sequence comparison (with or without gaps) is an $O(n^2)$ algorithm: on a sequential machine, time proportional to the square of the sequence length is required to solve the problem.[b] On the other hand, a linear systolic array with n processing elements can solve the problem in time *directly* proportional to the length of the input sequence. A full-fledged B-SYS system with five hundred processing elements could perform the sequence comparison problem over 500 times faster than its host. Projecting this to a 10 000-element sequence, not uncommon in the world of computational biology, the uniprocessor host (such as an Intel 80386) would require approximately 10 minutes, while a 212-chip B-SYS system (a more ambitious B-SYS implementation using current technology would require only 20 chips) would only need 0.06 seconds, about the time it takes to copy a database string from disk to memory. This factor of 10 000 speedup is crucial for the database applications of linear systolic arrays: B-SYS is envisioned as a filter which can quickly select a small number of interesting sequences from an entire database of known genetic information. The GenBank database (release 66.0, December 15, 1990) contains 51 306 092 bases (nucleotides) from 41 057 entries, while the PIR International Protein Sequence Database (release 27.0, December 31, 1990) contains 7 620 668 residues (amino acids) in 26 798 entries. Assuming sequence lengths of 1240 bases, an inexpensive 32-chip B-SYS co-processor could complete an exact search of the entire Genbank database in about 2.24 days, while on an Intel 80386 this search would require over 10 years (1.3 years on a Cray-2). Researchers can often prune the search space to a smaller collection of sequences (e.g., those from specific organism or chromosome), in which case a B-SYS system can perform many more comparisons in a much shorter amount of time, enabling quick experimentation with different similarity metrics,

Several sequence comparison problems, as well as their solutions on the Brown Systolic Array, will be considered in Chapter 7.

The combination of simplicity and size makes combinatorial problems ideal for systolic solution. Since there are many different variations on these problems, truly useful hardware must be easily programmable. As with all hardware, there is also a desire to get vast computing power for a very low cost. Systolic Shared Register architectures,

[b] Masek and Paterson have developpped an $O(\frac{n^2}{\log n})$ algorithm for strings of equal length from a finite alphabet with a minor restriction on the cost function. It will be faster for values of n greater than 263 000 and is not suitable for VLSI implementation.[116]

and the Brown Systolic Array in particular, are a perfect co-processor solution for these problems.

1.3 Contributions

This thesis presents a system designed to solve the inadequacies of both systolic hardware and systolic programming languages for a wide variety of applications. A close examination of existing systolic hardware and software systems will be seen to yield the following observations:

- Single-purpose systolic machines are too inflexible to be, in general, useful.
- Many arrays of simple processing elements include complicated message-passing circuitry, unnecessary for systolic algorithms, which raises the cost and complexity of the machine.
- Regardless of message-passing hardware, most systolic systems have excessively complicated methods of interprocessor communication.
- Many of these systems have ultra-powerful processing elements with very large amounts of memory. Although useful for some problems, many simple systolic algorithms do not need these resources. Thus, the extra power is often a waste of space and money.
- Many programming languages used with systolic systolic systems have problems similar to those of systolic hardware: their use requires complicated and convoluted thought.

The Systolic Shared Register architecture and the New Systolic Language, the major conceptual contributions of this thesis, address these issues. To meld ideas and application, the use of the Brown Systolic Array is also investigated.

1.3.1 A Class of Programmable Systolic Architectures

The Systolic Shared Register architecture eliminates the complicated inter-processor communication present in many machines. Processing elements are split into functional units and register banks. Adjacent functional units can thus communicate and compute using the shared registers. The four characteristics of the general SSR architecture are:

- Broadcast instructions.
- Regular topology.
- Systolic shared registers.
- A stream model of data flow.

As shall be seen, these attributes lead to simple and efficient systolic architectures for any topology or application.

1.3.2 A Systolic Array Implementation

Paper architectures are prevalent in the literature, but to gain a full appreciation of the Systolic Shared Register concept, a prototype system had to be built. The Brown Systolic Array, a linear SSR machine, was designed with the Magic VLSI design system developed at the University of California at Berkeley and extensively simulated with the Crystal timing analyzer and the COSMOS switch-level simulator.[118,129,14] The 6.9 mm × 6.8 mm 2 μm Complementary Metal Oxide Silicon (CMOS) chip contains 85 000 transistors forming 47 processing elements. It was fabricated by the MOS Implementation Service (MOSIS), and the 94-pin chips were received in the late May of 1990. Of the twenty-four chips delivered, ten performed satisfactorily, and were combined to form a 470-processor array.

The prototype board for the 470-processor array was built for a personal computer with an Intel 80386 central processing unit (CPU). The resulting array is able to, for example, compare 470-character sequences over 80 times faster than the host. A custom-built board, such as that proposed in Section 5.2.4, could easily increase this to 600 or more times faster than the host using the same number of B-SYS chips.

The Brown Systolic Array successfully balances programmability, complexity, and cost to provide a powerful systolic co-processor for computational biology and other applications.

1.3.3 Software Fault Detection for Systolic Arrays

Any massively parallel machine must be resilient in the face of faults. When thousands or millions of processing elements are being used, a random fault at one processing element could corrupt an entire calculation. The simplistic method of fault detection is to run the entire computation twice and compare the two results. Although this method will often detect faults, it relays *no* information about where the fault occurred, allowing no array reconfiguration or software modification to avoid the faulty area. The solution to this problem is to check the computation periodically at many (or all) locations throughout array. Hardware fault detection methods rely on the duplication of memory and processing elements to mirror a computation. Unfortunately, these methods are fixed at implementation time: if more (or less) fault protection is desired, changes cannot be made. For example, many matrix operations can be made fault tolerant by adding checksum rows and columns.[66] Ideally, when such operations are being performed the processing power previously allocated to fault detection should be reallocated to computation. This thesis presents the method of *software fault detection*, an ideal solution to the fault detection problem for programmable systolic arrays and SSR machines in particular.

1.3.4 Programming Methods for Systolic Arrays

Systolic programming languages generally fall into two categories: those which are abstract and those which are not. The former rely on mathematical mappings to define the systolic data movement and a separate program to specify the cell function. The latter generally place a communications burden on the user: unnatural instructions must be used to move data between processing elements and deadlock is always a concern. Following from the idea of using data streams for systolic programming, the New Systolic Language (NSL) is introduced. NSL combines the simplicity of inline control with the elegance of the abstract methods. The author has written a prototype NSL system in C++ to generate B-SYS code.

1.4 Overview of this Thesis

After this introduction follows a chapter analyzing the needs of a massively parallel yet inexpensive systolic array for combinatorial problems. Chapter 2 investigates previous work concerning all three parts of the systolic co-processor triad: algorithms, architecture, and programming languages. Section 2.1 further describes the concept of the systolic algorithm and considers an illustrative selection of systolic applications. The requirements of these applications guided the the Brown Systolic Array and New Systolic Language implementations. Section 2.2 examines several systolic and semi-systolic systems and then briefly summarizes a sampling of other hardware. The SSR architecture capitalizes on their features and avoids their drawbacks. Section 2.3 discusses and analyzes the current state of systolic programming, forming the basis for the New Systolic Language.

The class of Systolic Shared Register architectures is defined in Chapter 3. Chapter 4 details the B-SYS implementation: its architecture, design, and simulation. Chapter 5 considers the use of the B-SYS chips: fault detection and testing, board design, and prototype performance. Chapter 6 presents the NSL framework for systolic programming. Chapter 7 takes a close look at several applications on B-SYS, in particular several sequence comparison problems important to the Human Genome Project. A review of the B-SYS integrated systolic system concludes this thesis in Chapter 8.

Appendix A is a user's guide to the B-SIM simulator of the Brown Systolic Array. Appendix B describes the use of the B-SYS prototype hardware. Appendix C considers sequence comparison on the Connection Machine.[c]

[c] Connection Machine and C* are registered trademarks of Thinking Machines Corporation. CM-2, CM, and Paris are trademarks of Thinking Machines Corporation.

Chapter 2

Related Work

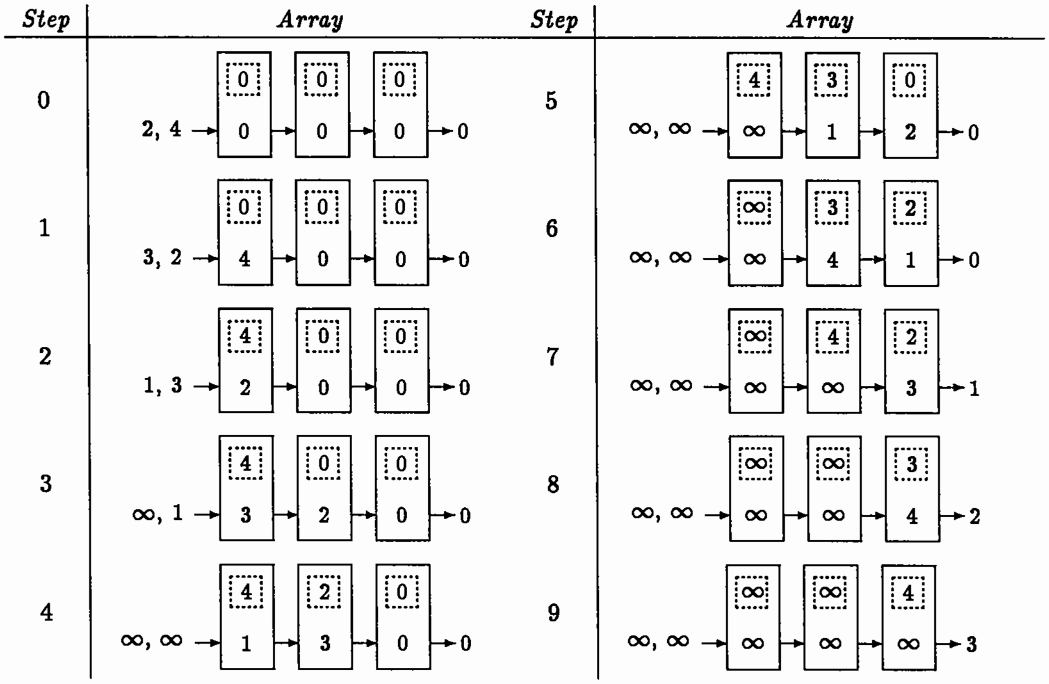
THIS chapter explores the three arms of systolic system design: algorithms, architectures, and programming languages. Previous work in each of these areas is evaluated, setting the stage for the introduction of the Systolic Shared Register architecture and the New Systolic Language. This critical review is vital to a complete understanding of SSR architectures and the Brown Systolic array.

2.1 Systolic Algorithms

This section further examines the systolic paradigm, starting with a simple systolic sorting algorithm later used to illustrate programming techniques on SSR machines. Another canonical example, a more computational problem, is Horner's method of polynomial evaluation. Programs for this problem in several systolic programming languages are discussed in Section 2.3. Sequence comparison, a dynamic programming algorithm of particular importance to the biological applications of the Brown Systolic Array, is also reviewed. Several variations on this fundamental problem, and their B-SYS programs, are analyzed in Chapter 7. Finally, a collection of systolic algorithms from computational geometry, cryptography, matrix algebra, and signal and image processing are summarized with an eye toward their respective hardware requirements.

2.1.1 Sorting

The simplest systolic sorting algorithm uses a linear array of processing elements. The function of each processing element is to read a number from the left, compare it with a stored value, and pass the smaller to the right. In this way, the smaller values will exit the array (from the rightmost cell) in sorted order. Using this algorithm, a type of parallel insertion sort, $n+1$ numbers may be sorted on n processing elements in $3n+1$ steps. Each processor is assumed to have a local storage register and a scratch register. During each step, the processors will compare the number in the scratch register with the stored number and then store the larger of the two numbers and pass the smaller to the right. Several steps of this sorting algorithm are shown in Figure 2.1, wherein the top register is the local storage and the bottom is the scratch register. The array is



The local storage register is enclosed in a dotted box. The lower scratch register is used for communication. The array is shown after each complete step, comprised of a comparison phase and a data movement phase.

Figure 2.1: Systolic insertion sort of 4, 2, 3, 1.

shown after the completion of each comparison and movement step. For example, during step 3 the values 4 and 2 are compared in the first processor. The value 2 is passed to the second while 4 is retained. The input value 3 is passed to the first processor at the same time it passed 2 to the second.

This example illustrates all the important principles of a systolic algorithm. Data moves through the array in a regular, synchronous fashion. All data movement is between adjacent processing elements. Finally, the basic operation of each cell is quite simple. For this reason, the replication of the basic cell to form a very large systolic array is easy using current technologies.

2.1.2 Polynomial Evaluation

The systolic implementation of Horner's method for evaluating a degree d polynomial illustrates the use of a systolic pipeline similar to that several digital signal processing algorithms.[59,138] The $d+1$ coefficients are stored in the linear array of processors, while successive x values are passed through the array for evaluation. The polynomial evaluation is broken down into the familiar

$$y_i = c_0 + x_i(c_1 + x_i(c_2 + x_i(c_3 + x_i(c_4 + \ldots)))). \tag{2.1}$$

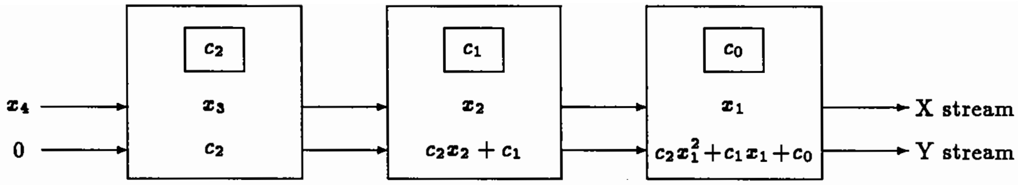


Figure 2.2: Systolic implementation of Horner's method.

Thus, a polynomial of degree d may be efficiently evaluated with $d + 1$ multiplications and $d+1$ additions. Equation 2.1 can be expressed with the following FOR loop program:

```
FOR $j = 1$ TO $n$
    FOR $i = d$ DOWNTO 0
        $y_j = c_i + x_j y_j,$
```

or recursively as:

$$c_{i,j} = c_{i,j-1} \tag{2.2}$$
$$x_{i,j} = x_{i-1,j} \tag{2.3}$$
$$y_{i,j} = x_{i,j}y_{i-1,j} + c_{i,j}. \tag{2.4}$$

The variables from Equation 2.1 have been expanded to be indexed by both loop indices. Note, however, that the c values remain invariant over changes in j and the same holds true for x over changes in i. The initial conditions for this recurrence are:

$$c_{i,0} = c_{d-i} \tag{2.5}$$
$$x_{0,j} = x_j \tag{2.6}$$
$$y_{0,j} = 0, \tag{2.7}$$

where c_{d-j} and x_j refer to values in the original equation. The c values have been reversed for simplicity.

A number of processing elements equal to one more than the degree of the polynomial is used. This simple method is typical of many systolic programs: each processing element executes the same instructions (a multiply and an addition). The systolic algorithm is shown pictorially in Figure 2.2 and, as mentioned, programs for this algorithm are analyzed in Section 2.3.

2.1.3 Sequence Comparison

Sequence comparison, a primary application for the Brown Systolic Array, is the problem of determining the similarity of two (or more) sequences. The dynamic programming solution for basic sequence comparison problems is a combinatorial algorithm, involving only integers and simple operations. For general information on sequence comparison and its variations, the reader is referred to the literature.[4,135,156]

Let the two strings being compared be $A = a_1a_2\ldots a_n$ and $B = b_1b_2\ldots b_m$ with each $a_i, b_j \in \Sigma$, where Σ is a finite alphabet. The distance between two strings A and B is denoted as $\text{dist}(a_1\ldots a_n, b_1\ldots b_m)$, or $d_{n,m}$ for short, and refers to the minimum cost sequence of single-character edit operations (insert, delete, and transform) required to transform A to B. For example, with an insertion and deletion cost of 1 and transformation cost of 2, the distance from the string AACUG to ACCUGA is 3: the mutation of the second A to C, and the addition of the final A.

Dynamic programming can be used to calculate edit distances by the extension of partial homologies, or substring matchings. To calculate $d_{i,j}$, the best of the three possibilities (add, delete, and transform a_i to b_j) is picked. The recurrence is:

$$d_{0,0} = 0 \tag{2.8}$$

$$d_{i,0} = d_{i-1,0} + \text{dist}(a_i, \phi) \tag{2.9}$$

$$d_{0,j} = d_{0,j-1} + \text{dist}(\phi, b_j) \tag{2.10}$$

$$d_{i,j} = \min \begin{cases} d_{i-1,j-1} & + \text{dist}(a_i, b_j) \\ d_{i,j-1} & + \text{dist}(\phi, b_j) \\ d_{i-1,j} & + \text{dist}(a_i, \phi) \end{cases} \tag{2.11}$$

The empty string and the null character are both denoted as ϕ and it is convenient to assume that each string begins with a null character. In the initial conditions, $d_{i,0} = \text{dist}(a_1\ldots a_i, \phi)$ is the cost of deleting $a_1\ldots a_i$ to transform it to the empty string ϕ. The first case of Equation 2.11 is matching the character a_i to b_j, and then adding the cost of matching the first $i-1$ elements of A to the first $j-1$ elements of B. The second case is inserting a character into A to make A equal to B, then matching the first i elements of A to the first $j-1$ elements of B (in effect, the inserted character is matching b_j). The final case is the deletion of a character from A to make it equal to B, and is symmetric to the previous case.

The dynamic programming solution to Equation 2.11 uses the string A to index the rows of a table and the string B to index the columns. The elements of the table are then entered according to the definition of d with the table entry (i, j) corresponding to $d_{i,j}$. The dynamic programming algorithm completes the table as the data dependencies allow (for example, $d_{i,j-1}$ must be calculated before $d_{i,j}$). The reason this works is that if the minimum distance $d_{m,n}$ between the two strings matches a_i to b_j, then the distance calculation must use the minimum matching of $a_1\ldots a_i$ to $b_1\ldots b_j$, for if not $d_{m,n}$ has not been calculated as a minimum.

The comparison of the two strings AAC and AGCA will result in the table of Figure 2.3 using the dynamic programming method. The cost of matching the two strings is the final (boxed) table entry, $d_{3,4} = 3$. Note that the cost of matching AAC to AGC can also be read from this matrix as $d_{3,3} = 2$. The actual sequence of edit operations used to make the "evolutionary change" may be kept track of as the matrix is calculated if such information is needed.

There are many different ways to map this computation to a systolic array.[a] Perhaps the simplest method is to assign one processing element to each column of the dynamic

[a] Section 2.3 will discuss the problem of systolic mappings.

	ϕ	A	G	C	A
ϕ	0	1	2	3	4
A	1	0	1	2	3
A	2	1	2	3	4
C	3	2	3	2	$\boxed{3}$

Figure 2.3: Comparison of AAC with AGCA.

programming matrix (Figure 2.3). To initialize the computation, the first string is loaded into the array, one character per processing element. Then, the second string is shifted through the array along with weight values. Thus, $d_{i,j}$ will be computed in processor $\mathcal{P}_j$ at time $(i + j)$. The values necessary to complete this computation ($d_{i-1,j}$, $d_{i,j-1}$, and $d_{i-1,j-1}$) have already been calculated when it comes time to determined $d_{i,j}$: $d_{i-1,j}$ and $d_{i,j-1}$ were calculated during the previous step in processors $\mathcal{P}_j$ and $\mathcal{P}_{j-1}$, respectively, and $d_{i-1,j-1}$ was calculated in $\mathcal{P}_{j-1}$ two time steps ago. As can be seen, this data movement maintains the systolic property that data needed for computation in a processing element is computed near that processing element in both time and space.[b]

Other Variations

There are many variations on the sequence comparison problem, including sequence to subsequence comparison, longest common subsequence comparison, and the addition of affine gap penalties to the distance metric. For example, in the DNA case it is more likely that the elimination of 10 consecutive nucleotides is the result of a single event (a large cut) than the result of 10 independent events (each removing one of the nucleotides in the subchain). Judicious use of gap penalties will mirror this behavior in the cost function. Sankoff and Kruskal discuss algorithms for several other sequence comparison problems.[135] A previous paper by this author considers the problem of comparing a nucleic acid to codings of an amino acid sequence using the genetic code.[68]

Considering the underlying encoding of the amino acids shown in Table 1.1 (page 3), some amino acid mutations (e.g., phenylalanine to leucine) are more likely than others (e.g., phenylalanine to arginine). Also, biologists consider the occurrence of some amino acids to be more important than others: in actual proteins, asparagine mutates approximately seven times more often than tryptophan. Thus, to detect significant alignments between proteins, a cost function dist(a, b) using different values for each possible mutation is required. The most common metric used is that proposed by Dayhoff.[32,137] Without a *programmable* systolic array, the usefulness of different metrics and methods cannot be quickly evaluated.

Sequence comparison in all its varieties forms the basis of many data analysis problems involving biological sequences, speech processing, polymer analysis, and image processing. Several sequence comparison problems are considered in Chapter 7.

[b] This mapping will be covered in more detail in Section 7.1, where a B-SYS program for the computation is presented.

Problem	EC	Data	CPU[a]	Tp[b]	S[c]	C[d]	T[e]	P[f]	N[g]	M[h]	Notes
Alternate Transcription[68]	2	int	add	L	n	n^2	n	n	n	1	Primary application is amino acid encoding regions in DNA.
Articulation Points[11]	1	int	add	M	n^2	n^3	n	n	n^2	1	Also has bridges, shortest cycle, MST, cyclic index, and bipartite graph.
Cartesian Product[101]	1	any	cmp	L	n	n^2	n	n	n	1*	Same mapping: join operations, matrix vector multiplication, triangular linear systems. *$O(n)$ I/O ports.
Channel Assignment[71]	1	int	add	L	n	$n \log n$	n	n	n	1	n = wires.
Constrained Least Squares[140]	2	float	div	T	n	n^2	n	n	n	1	
CFG Parsing[24]	1	char	cmp	M	n	n^2		n^2	1	1	Approx. string matching, string-to-string correction, longest common subsequence.
Convex Hull[74]	1	int	add	M	n	n^2	n	1	n^2	1	Also has a 3D algorithm.
Convex Hull[25]	1	float	add	L	n	n^2	n^2	1	n	1	Incremental algorithm. Two different arrays are presented, one for add, delete, hull and the other for add, clockwise hull.
1-D Convolution[93]	1	float	mult	L	n	n	n	1	n	1	n is size of convolution.
2-D Convolution[43]	1	any	mult	M	nm	n^2m^2		m^2	n^2	1	Image correlation, object labeleing, binary erosion, binary dilation, gray scale erosion, gray scale dilation over various opererations in place of add and mult.
Data Compression[146]	1	char	cmp	L	kn	kn		1	k	1	
1-D Deconvolution[101]	1	float	mult	L	n	n^2	n	n	n	1	Same mapping: correlation, polynomial multiplication and division.
1-D DFT[101]	1	float	mult	L	n	n^2	n	n	n	1	
Distance Graph[93]	1	int	add	H	n^2	n^3	n	n	n^2	1	Matrix multiplication over + and min.
Distance Graph[102]	1	int	add	L	n^2	n^3	n^2	n^2	n^2	1	Matrix multiplication over + and min.
Inner product[97]	1	any	mult	L	n	n^2	n	n	n	1	
1-D FFT[148]	2	float	mult	L	n	n^2	n	n	n	3	Radix 4 pipeline. Delays up to 768 elements make large memory requirement.
FIR Filter[23]	1	int	mult	L	n	nK	n	1	K	1	K-tap finite impulse response: $y_n = \sum_{k=0}^{K-1} a_k x_{n-k}$.
IIR Filter[23]	2	int	mult	L	n	nK	n	1	K	1	K-tap infinite impulse response: $y_n = \sum_{k=0}^{K-1} a_k x_{n-k} + \sum_{k=1}^{K-1} b_k y_{n-k}$. See also Knowles on bit-level pipelined IIR filters.[81,82]
Kalman Filtering[57]	2	float	sqrt	M	n	n^3	n	n	n^2	1	Outputs reused as inputs
Long Multiplication[101]	1	int	mult	L	n	n^2	n	n	n	1	
LU Decomposition[94]	2	any	recip	H	n^2	n^3	n^2	$4n$	n^2	1	
Matrix Inversion[40]	4	any	mult	M	n^2	n^3	$5n$	$5n$	$2n^2$	1	

[a] Most complicated operation in algorithm. [b] Topology: Linear, Mesh, Triangular mesh, Hexagonal mesh. [c] Problem size, in terms of n. [d] Sequential complexity of the obvious sequential algorithm, in terms of n. [e] Parallel time, in terms of n. [f] Parallel period, in terms of n. [g] Array size ($\propto A$ (Area)), in terms of n. [h] Amount of memory required per processor (order of magnitude: $\leq 10^M$ words are required).

Table 2.1: Systolic algorithms.

Problem	EC	Data	CPU	Tp	S	C	T	P	N	M	Notes
Matrix–vector multiplication[94]	1	any	mult	L	n, n^2	n^2	n	n	n	1	Numbers for storing matrix in array for multiple computations. Requires $O(n)$ I/O ports.
Matrix Multiplication[93]	1	any	mult	H	n^2	n^3	n	n	n^2	1	
Matrix Multiplication[102,133]	1	any	mult	L	n^2	n^3	n^2	n^2	n^2	1.1	
Matrix Transposition[128]	1	any	move	M	n^2	n^2	n	n	n	1	
Minimum Spanning Tree[39]	1	int	add	L	n	n^2	$\frac{n^2}{p}$	$\frac{n^2}{p}$	p	$\frac{n^2}{p}$	MST on n nodes and p processing elements.
Nearest Neighbor[107]	2	float	square	L	n	n^2	n	1	n	1	Based on key enumeration algorithm also presented.
Optimal parenthesization[61]	1	int	add	T	n	n^3	n	n	n^2	1	RNA folding, see also work of Moldovan.[122]
Order k 1D ROS[45]	1	any	cmp	L	n	$\log n$	n	1	k	1	kth Median Filtering, based on priority queue (ROS = running order statistic)
Polygon Intersection[25]	1	float	add	L	n	n^2	n	1	n	1	Also has solutions for various inclussion, intersection, and closest-point problems.
Polynomial evaluation[139]	1	any	mult	L	n	n	n	1	n	1	n is polynomial degree.
Polynomial GCD[19]	1	int	div	L	n	n^2	n	n	n	1	Used for Reed-Solomon error correcting codes.
Polynomial Multiplication[88]	1	float	mult	L	n	n^2	n	n	n	1	Also polynomial division
Priority Queue[104]	1	any	cmp	L	n	$\log n$	n	1	n	1	
Projection[76]	4	float	sqrt	T	n	n^3	n	n	n^2	1	QR decompostion via Givens rotations used.
Relational Database[92]	2	any	cmp	M							Pipelines for comparison, intersection, duplicate removal, union, projection, join, equi-join, relational division
Relational Database[103]	1	any	cmp	M	n	kln	n	n	kl		For l queries of size k on a database of size n. PEs need to store simple programs containing the desired queries.
RSA Cryptography[164]	1	MP[a]	mod	L	n	n	nT_n	T_n	n	3	n is bits, T_n is time for modular multiplcation of $n > 300$ bits.
Searching[67]	1	any	cmp	L	n	$\log n$	n	1	n	.5	
Sequence Comparison[109]	1	int	add	L	n	n^2	n	n	n	.5	Has versions for strings and substrings, with and without gap penalties.
Shuffle Scheduling[71]	1	int	add	L	n	n^2	n	n	n	1	n = nodes.
Sorting[150]	1	any	cmp	L	n	$n \log n$	n	n	n	.5	A systolic bublesort. Non-optimal AT^2.
Stereo Matching[60]	1	int	add	L	n, m	nm	n	m	n	1	For n right features and m left features
Transitive Closure[61]	1	bool	logical	M	n	n^3	$5n$	$5n$	n^2	1	Matrix mult over logical AND and logical OR, n = nodes.
Transitive Closure[102,153]	1	bool	logical	L	n	n^3	n^2	n^2	n^2	1.1	

Other algorithms include: Back substitution, 2-D Comparison, 1-D Correlation, 2-D Correlation, 2-D DFT, Discrete Hadamard Transform, Edge Detection, Eigenvalue-Decomposition, 2-D FFT, Gauss-Seidel Iteration, 2-D Gray Scale Normalization, 2-D Hadamard Transform, Jacobi Iteration, Linear Phase Filter, QR Decomposition, Singular Value Decompostion, Toeplis Systems, Unconstrained Least Squares, Wiener Filtering, and Z-transform. See also the book by S. Y. Kung[97] and the surveys of Li et. al.,[106] and Manohar.[115]

[a] Multiple-precision integer.

Table 2.1: Systolic algorithms. (cont)

2.1.4 Other Algorithms

Table 2.1 summarizes a selection of systolic algorithms. For each algorithm, several items are listed: the number of types of processing elements (equivalence classes), the type of data used (integers, characters, or floating-point numbers), the complexity of the required CPU, and the topology of the array needed to perform the algorithm. Several expressions relating to the serial and parallel complexity of each algorithm are also tabulated.

The ability to support every one of these algorithms would be a tremendous burden on the size of individual processing elements, and hence the possible amount of parallelism in the final array. Examining the table, one notes that very few algorithms require more than one type of processing element, thus the broadcast of identical instructions to all elements of the array will be a reasonable method of control. Some algorithms with multiple equivalence classes, such as the alternate transcription problem for amino acids (Section 7.1.4), can be programmed with no instruction overhead because one cell program is a restriction of the other. The more complicated algorithms (constrained least squares, 1-D FFT, IIR, etc.) can still be executed by alternately turning on each equivalence class of processing elements to simulate multiple instruction streams.

On consideration of the presumed users of the Brown Systolic Array (in particular, computational biologists), complex arithmetic units may also be trimmed: the lack of floating-point arithmetic, division, and root extraction will eliminate the possibility of running several signal processing algorithms, however it will *add* an enormous amount of processing power to the final system since the chips will contain many more processing elements. Integer multiplication and division is not included for similar reasons, though of course products can be easily computed with addition.

Since B-SYS was initially designed as a programmable sequence comparison engine, a linear systolic array was the topology of choice. As can be seen from Table 2.1, a large number of applications run on linear systolic arrays, including some which seem inherently two-dimensional. For example, transitive closure, a type of matrix multiplication, can be performed on B-SYS. Although the algorithm (discussed in Section 7.2) provides only a factor of n speedup (versus the mesh algorithm's n^2), it illustrates the versatility of the linear systolic array. Although in the ideal world one would have programmable systolic co-processors of all topologies and sizes, in the real world a single linear SSR machine will often be both practical and sufficient.

Finally, the majority of the algorithms shown do not require large amounts of memory: this wasteful feature may also be discarded.

Pruning the list of applications, the following are appropriate for the B-SYS implementation: alternate transcription, channel assignment, data compression, distance graph, inner product, minimum spanning tree, priority queue, searching, sequence comparison, shuffle scheduling, sorting, stereo matching, and running order statistic. Several other applications can be executed on linear SSR machines like B-SYS, though not as efficiently because they require multiplication. These include: convex hull, 1-D convolution, 1-D deconvolution, inner product, 1-D FFT, FIR filter, IIR filter, long multiplication, matrix multiplication, nearest neighbor, and polygon intersection.

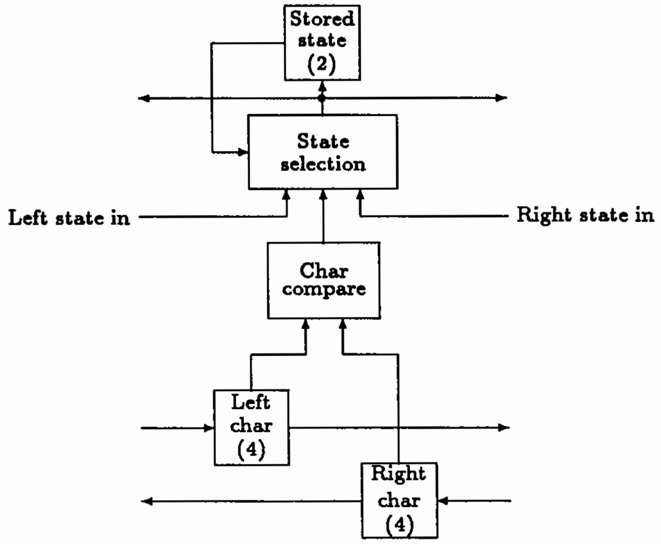


Figure 2.4: P-NAC architecture.

Although, as shall be seen, the Systolic Shared Register concept can be used with any systolic topology, B-SYS is a linear array which fills the niche of solving many combinatorial problems.

2.2 Systolic Hardware

The design of the Brown systolic array has been influenced by several systolic and semi-systolic architectures. This section examines a number of architectures which represent important milestones in the landscape of massively parallel machines. After considering a single-purpose machine, a massively parallel machines, and a powerful systolic multiprocessor, the tapestry is filled out with a summary of over twenty machines. Many novel features present in these machines have been adapted to the B-SYS architecture, while several of the more cumbersome features have been avoided. The specifics of the Brown Systolic Array implementation follow directly from an examination of the attributes of both the machines in this section and the algorithms presented in the previous section.

2.2.1 The Princeton Nucleic Acid Comparator

The Princeton Nucleic Acid Comparator (P-NAC) was developed by Daniel Lopresti and is the direct predecessor of the Brown Systolic Array.[109,110] P-NAC is designed to solve the sequence comparison problem over a four-character alphabet. Using the simple dynamic programming method of calculating edit distances (Section 2.1.3), P-NAC provides optimal asymptotic speedup by performing $O(n^2)$ computations in $O(n)$ time using $O(n)$ processing elements. The processing elements are connected in a linear array, and data is fed in from both ends of the array. Each processor is small — the 1985 nMOS implementation placed 30 processors on a 4.6 mm × 6.8 mm chip with a feature size of 4 μm ($\lambda = 2$)[c] — and the resulting chip can compare nucleotide sequences quickly. For example, a 1999-processor array of P-NAC processors can compare two 1000-character strings in 130 ms. The architecture of the P-NAC basic cell is shown in Figure 2.4.

Unfortunately, P-NAC can solve only a single problem. Even slight variations to the sequence comparison problem, such as the use of different costs in the calculation of the edit distance, cannot be solved with P-NAC. Since building a new chip for each application is clearly inefficient, a programmable successor to P-NAC is needed.[d]

Because only one algorithm is ever executed on P-NAC, systolic communication is implemented with latches and fixed connections. A special-purpose implementation of the sorting algorithm of Section 2.1.1 would use a similar method of communication. A programmable systolic array should, if possible, include a method of systolic communication which is as simple as the intrinsic communication of these special-purpose architectures.

A programmable systolic array should preserve the vast parallelism easily attainable with P-NAC. Since nucleic acids such as DNA can be thousands of nucleotides long, thousands of processing elements can be used effectively to accelerate database searches for similar pairs of sequences. Without small and simple processors, such as those of the P-NAC system, the cost of such huge arrays would be prohibitive.

2.2.2 The Connection Machine

The Connection Machine features 64 K (65 536) single-bit processors and a hypercube routing scheme.[152] Each Connection Machine processor (CM-2) can access 64 K bits of memory and a floating-point unit shared by thirty-two processing elements. The hypercube routing network provides arbitrary communication paths between processing elements. The four flag registers (see Figure 2.5) are available to neighboring and distant processing elements via the chip's router. The Connection Machine features "data level parallelism", which is to say that it is an SIMD (single instruction and multiple data stream) machine.

Although the Connection Machine supports non-systolic algorithms, it can also efficiently execute systolic algorithms intended for a square mesh or linear array of processing elements. Communication on the Connection Machine is, however, much less intuitive than the synchronous communication of single-purpose systolic arrays: the data must be explicitly moved through the array using the communication registers.

It might also be pointed out that bit-serial operations are slower than bit-parallel when data close to (but below) the computer's word size is being manipulated. In the

[c] Feature size and lambda-based design rules are defined in Section 4.2.1.

[d] The University of North Carolina at Chapel Hill is developing BioSCAN, a partially programmable biosequence comparison system based on a 2000-processor 0.8 μm CMOS chip. Although able to processes different cost functions over a 20-character alphabet (the amino acids), it is unable to perform more complicated versions of sequence comparison such as those with insertions and deletions or gap penalties, and thus is still restricted in application.[35,142] BioSCAN's algorithm is discussed in more detail in Section 7.1.

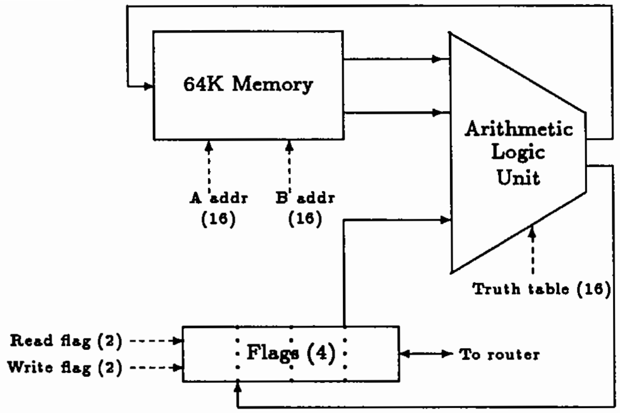


Figure 2.5: Connection Machine processor architecture.[63,152]

sequence comparison case, approximately ten times as many instructions are required on a bit-serial machine as on a bit-parallel machine with an eight-bit word. The use of bit-serial processing elements is partly the result of VLSI packaging restrictions: a chip with a large two-dimensional array of processing elements will likely not have enough pins to allow word-parallel input and output (I/O) at the chip boundaries. In fact, some chips use a smaller word size for chip I/O than for internal processing.[54] This problem is not present in linear systolic arrays since they only communicate through the two end processors.

2.2.3 Warp

The Warp computer was developed at Carnegie Mellon University under the guidance of H. T. Kung, and is a powerful systolic multiprocessor.[8] The present model, PC-Warp, consists of ten 10-MFLOPS (million floating-point operations per second) processors connected in a linear array. The array cells are independently programmable, each containing memory for 8 K instructions (thus, Warp is an MIMD, or multiple instruction and multiple data stream computer). Each Warp processing element features a 32-bit multiplier, a 32-bit adder, and 32 K words of local memory. Between neighboring processors there are three communication queues able to store 512 32-bit words each. The components of the processor are connected with a 32-bit crossbar switch. Thus, data may be placed on the queues directly from the units of CPU or from the processor's local memory. Warp provides hardware control of the queues: a full or empty queue will cause the appropriate processor to wait until this condition changes. Figure 2.6 shows the data path of an individual Warp cell. A new version of Warp, called iWarp, is being developed jointly by Carnegie Mellon and the Intel Corporation and will have 64 processors and more memory per processor.[18]

The Warp system can perform a large number of signal and image processing applications quickly.[36] However, it is not especially suited for symbolic or simple computations

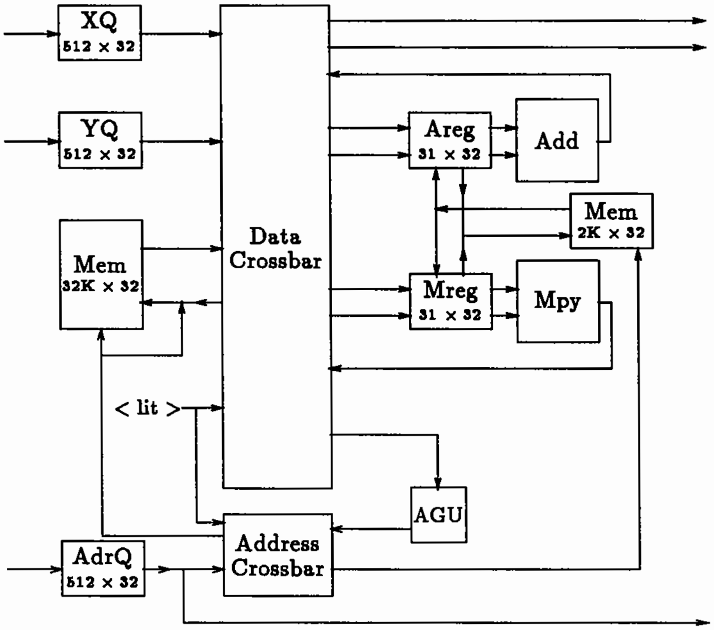


Figure 2.6: Warp data path.[8]

on large amounts of data. Since the size of the array is limited by the size of the processing elements (one board each), Warp is unable to make full use of the vast parallelism inherent in many problems.

2.2.4 Other Hardware

A systolic building block is a chip (or board) which, when replicated, can be used to build powerful systolic arrays. As with complete systems, not all systolic building blocks have been made into arrays, but their architectures are of interest. This section briefly discusses several other systems which influenced the design of the Brown Systolic Array.

Splash

Splash, developed at the Supercomputer Research Center in Bowie, MD, can be best described as a *firmware* programmable systolic array.[58] It is a linear array of 32 Xilinx field-programmable gate arrays (FPGAs)[161] with memory chips in between. Each FPGA has a 20 × 16 mesh of configurable one-bit logic blocks and 144 input/output blocks. The FPGAs also have sophisticated routing facilities for local and global connections. The FPGAs are used as programmable hardware: a program of connections between and functions for the logic blocks is preloaded into the array. As data is pumped through the array, it is processed according to the preloaded configuration. Simple applications, such as the basic sequence comparison problem on a 4-character alphabet, can be executed nearly as quickly as on comparable single-purpose hardware (see Section 7.1.2).

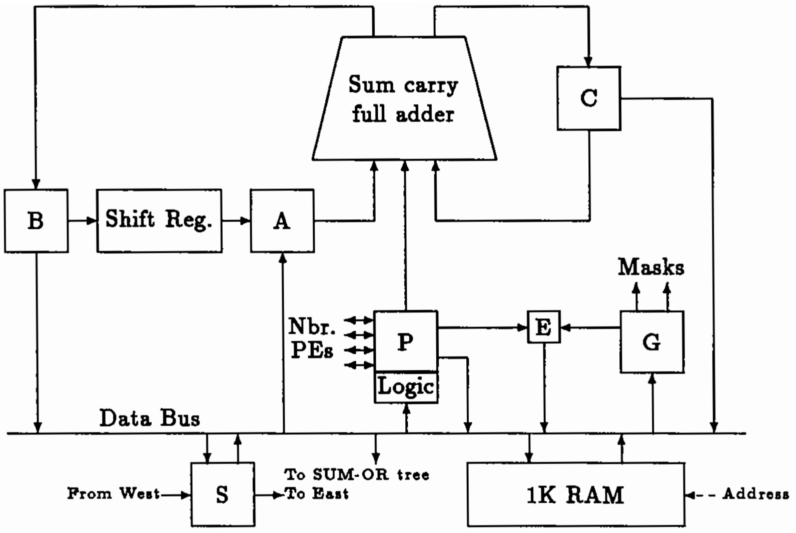


Figure 2.7: Massively Parallel Processor architecture.[13]

There are, however, two main problems with using the Splash system for general systolic algorithms.[112] First, slight variations in algorithm require that the lengthy programming task be redone. Second, Splash implementations have a low processor density per FPGA. Splash is well suited for problems involving small words (2-bit words suffice for nucleotide sequence comparison costs), but performance degrades considerably as the word sizes increase. Of the algorithms which have been implemented on Splash, typically eight or sixteen systolic cells have been placed on each chip, a marked contrast to B-SYS's 47 cells per chip, especially when one notes that B-SYS does not push the limits of current VLSI technology as the Xilinx chips do. Still, for appropriate applications Splash performs very well and, in spite of programming difficulties, can be reconfigured for new applications ll sor faster than single-purpose VLSI chips can be fabricated.

Image Processing

Simple mesh architectures of bit-serial processors (i.e., the Connection Machine without its extensive routing) have been used for many image processing applications. The Massively Parallel Processor developed by Goodyear and NASA stands out as an early example (Figure 2.7).[13,132] T. J. Fountain has surveyed several SIMD bit-serial image processing systems.[53] Each of the eight machines surveyed has between 8 and 64 processors per chip. The NCR Geometric Arithmetic Parallel Processor (GAPP) also falls in this class of bit-serial mesh architectures.[125]

Digital Signal Processing

Digital signal processing (DSP) generally requires a full range of floating-point operations. Among these more powerful systolic processors is the Programmable Systolic Chip

(PSC) developed at CMU.[48] The PSC featured simpler processors than those used with Warp (an 8-bit word, but still including a multiplier), and was designed for use in linear arrays as well as various other topologies. Other powerful programmable systolic arrays have been constructed using the INMOS Transputer and other DSP chips,[65,96] as well as custom-designed processors.[51,126]

The Instruction Systolic Array

The proposed Instruction Systolic Array (ISA) is a mesh of simple processing elements in which *both* data and control flow through the array.[87,100] Instructions flow in one direction (north to south) and selectors (or mask bits) flow in the other (west to east). The ISA thus uses a clever solution to the problem of controlling n^2 processing elements with only n control bit inputs each time step. The first proposal of the Brown Systolic Array considered the use of systolic instructions in a linear systolic array.[69] However, the problems with this concept probably outweigh the benefits in a linear systolic array.

Hardware Summary

Table 2.2 summarizes information about the systolic building blocks described in this chapter as well as in the literature. For each building block, those figures of merit which could be gleaned from the reference are tabulated. As can be seen, the Brown Systolic Array (with its modest number of transistors) has a high number of bit-parallel processing elements on each chip: a tribute to the elegance of the Systolic Shared Register design.

Also, B-SYS has a relatively small amount of local memory and no off-chip memory. The advantages of this are speed and size: lengthy memory accesses are avoided and a higher processing element density can be achieved. Although only a small number of registers is used, it is sufficient for many of the applications presented in Table 2.1.

The length of the B-SYS word falls between the extremes (bit-serial and 32-bit floating-point). The clock speed is typical of many of the building blocks, and both the clock speed and the speed of individual processing elements could be improved with a new design. Floating-point arithmetic is, as mentioned, not needed for B-SYS's target applications.

Table 2.3 tabulates information about complete systems: the total number of processing elements and memory, array topology, and system size. (Splash has not been included because of its variable number of systolic cells.) A full-fledged B-SYS system with 1504 processing elements could provide a large computational power (3 or more billion operations per second) using very little space or, equivalently, at very low cost. Thus, B-SYS has one of the highest performance to cost ratio of the machines listed, a ratio higher than all general-purpose supercomputers (obviously, supercomputers can also crunch data on a wider range of problems than B-SYS).

It is important to remember that B-SYS, being a linear array, is eminently scalable: with nine bits of communication (plus clocks and possibly instructions), boards may be joined to form systems as large as desired, with only one or two boards requiring

Chip	Year	Technology	Transistors	PE per	Memory		Word	Clock	Speed	Floating-	ALU	Control
			×10^3	chip	on[a]	off[b]	(bits)	(MHz)		point?		
ICL DAP[49]	77	MSI		0.2	0	1	1			N	Add/And	SIMD
DTW[1,2]	81	3.5μm CMOS	5	1	7	0	16	5	5 MOPS	N	Add	MIMD
CLIP7A[54]	82	5μm CMOS	7	1	4	0	16(8)	5	1.2 MOPS	N	Arith	OC[c]
GEC GRID[10]	83	2.5μm CMOS	50	64	32	8	1	10	10 MIPS	N	Lookup	SIMD
MPP[13]	83	HCMOS	8000	8	36	1024	1	10		N	Lookup	SIMD
NTT AAP[83]	83	2μm CMOS	112	64	96	0	1	18	18 MIPS	N	Log/Add	SIMD
PSC[48]	83	4μm NMOS	25	1	54	0	8(9)	3		N	Mult	MIMD
CM-1[63]	85	2μm CMOSGA	10	16	8	4	1	4	4 MIPS	N	Lookup	SIMD
GAPP[127]	85	CMOS		72	128	0	1	10	10 MIPS	N	Arith	SIMD
ICL DAP-2[52]	86	CMOS	3	16	0	?	1		0.6 MOPS	N	Add/And	SIMD
Warp[6]	85	TTL		.004	0	4	32	5	10 MFLOPS	Y	Mult	MIMD
IMS T800-30[65]	86		150	1	4096	1024	32	20	2.5 MFLOPS	Y	Divide	OC
AIS-5000[136]	87	Gate Array		8	0	32	1	7		N	Arith	SIMD
CM-2[152]	87	2μm CMOS	14	16	4	64	1			OC	Lookup	SIMD
ASAP II[62]	88			16	1200	Y	1	20		N		SIMD
API15C[55]	89	CMOS		1	288	0	16	10	10 MIPS	N	Mult	SIMD
PCWarp[7]	87	TTL		.004	0	32	32	5?	10 MFLOPS	Y	Mult	MIMD
MasPar[160]	89	1.6μm CMOS		32	384	32	4[d]	14	1.8 MOPS	Y	Mult	SIMD
B-SYS	**90**	**2μm CMOS**	**85**	**47**	**16**	**0**	**8**	**12**	**2 MIPS**	**N**	**Arith**	**SIMD**
Blitzen[17]	90	1.25μm CMOS	1100	128	1024	0	1	20		N	BS	SIMD
iWarp[18]	90		600	1	128	1600	32	20	20 MIPS	Y	FP, M	MIMD
SLAP[47]	90	2μm CMOS	50	4	32	0	20	8		N	Mult	SIMD
ISA[87]	prj	CMOS		100	8	0	1			N		Systolic

[a] In words per processing element. [b] Addressing capability in K words. [c] Off chip. [d] Data path is 4 bits wide, but registers are 32 bit. Bit-serial communication to 8 nearest neighbors, MOP time is for 32 bit intruction.

Table 2.2: Systolic building blocks (by date).

Machine	Date	PEs	Mem (bits)	Class	Topology	Boards	Size	Speed	$\mathcal{D}$[a]
Warp[6]	85	10	1.3 M	MIMD	Linear	11	Medium	10 MFLOPS	IP
PCWarp[21]	87	10	10 M	MIMD	Linear	11	Medium	10 MFLOPS	IP
MICSMACS[55]	89	18	81 K	SIMD	Linear	2	Small	180 MIPS	GP
ESL[16,86]	81	20	320 K	SIMD	Linear	7	Medium	16 bit int mult ~ 200 MOPS	SP
Saxpy Matrix-1[51]	87	32	128 K	SIMD	Linear		Medium	1 GFLOPS	GP
iWarp[18]	90	64	256 M	MIMD	Toroid	~ 16	Medium	1.28 GFLOPS	GP
CLIP7A[54]	86	256	128 M	SIMD	Linear	140	Large		IP
SLAP[46]	90	512	0.3 M	SIMD	Linear	128 chips	Small		VP
ICL DAP[49]	77	1024	1 M	SIMD	Mesh	89	Large	Mat. mult. 32 x 32, 32FP: 16ms	SP
AIS-5000[136]	87	1024	32 M	SIMD	Linear	~ 10	Medium	8-bit Addition: 298 MOPS	IP
ASAP II[62]	88	1024		SIMD	Mesh	4	Med	32 Mult: 170 MFLOPS	GP
AMT DAP 510[143]	89	1024	32 M	SIMD	Mesh		Large	800 MFLOPS	IP
B-SYS	**90**	**1504**	**192 K**	**SIMD**	**Linear**	**1**	**Small**	**8-bit Addition: 3–5 GOPS**	**CP**
DTW[1,2]	81	1600	180 K	MIMD	Mesh		Medium	9-D feature throughput 100μs	SR
GAPP[125]	88	2304	288 K	SIMD	Mesh/Linear	~ 2	Small	8-bit Addition: 900 MOPS	IP
MPP[13]	83	16384	16 M	SIMD	Mesh	96	Large	8-bit Addition: 6.5 GOPS	IP
MasPar[160]	89	16384	2 T	SIMD	8Mesh/Xbar		Large	30 GIPS	GP
NTT AAP[83]	83	65536	6 M	SIMD	Mesh/Bypass	1024 chips	Large		IP
CM-1[63]	85	65536	32 M	SIMD	Mesh/Hypercube	128	Large	32-bit Addition: 2 GOPS	AI
CM-2[152]	87	65536	500 M	SIMD	n-Mesh/Hypercube	142	Large	8-bit Add: 4 GOPS; 20 GFLOPS[b]	AI

[a] Original problem domain: General Purpose, Signal Processing, Image Processing, Articicial Intelligence, Combinatoric Processing, Speech Recognition, or Video Processing.

[b] This anomoly is the result of performing the 8-bit addition in the bit-serial processing elements and the floating-point operations in the 2048 pipelined floating point units.

Table 2.3: Systolic systems (by size).

access to the system bus.[e] This is a great contrast to mesh networks, where large numbers of communication lines must be used to join smaller meshes together, and hypercube systems for which the original communication protocol of the system can limit the dimension (size) of the hypercube.

The examples of this section, from P-NAC to Warp, represent three schools of systolic thought: the simple single-purpose array, the SIMD array of bit-serial processing elements, and the MIMD array of powerful processors. The single-purpose architectures lack versatility while the MIMD machines lack massive parallelism. All three types of architecture present a specialized and not entirely satisfactory method of implementing systolic communication.

2.3 Systolic Programming Languages[f]

Having reviewed algorithms and architectures for systolic systems, it is now time to turn to the last part of the systolic co-processor triad: systolic programming languages. Although assembly language could certainly be used for all B-SYS applications, it is difficult and unwieldy for the casual user of the system. A close examination of several systolic programming languages will result in a list of criteria for the development of a new systolic programming language appropriate to the co-processor model of computation. The language should support different topologies, be part of a conventional language able to perform both host and co-processor functions and, perhaps most importantly, have an elegant method of programming different data flows.

In addition to motivating the introduction of the New Systolic Language (Chapter 6), this section continues to explore the systolic paradigm.

2.3.1 Language Types

In the torrent of papers on systolic solutions for divers applications, the twin problems of systolic programming and generalized systolic design have received somewhat less attention. The problem of systolic programming refers to building languages or subroutine libraries for general-purpose systolic processors or multiprocessors: providing methods to program an existing machine. On the other hand, generalized systolic design systems aid in mapping an algorithm to the systolic domain. Using such tools, various interconnection networks and data flows may be explored as a precursor to building or programming a systolic array. These systems are especially useful to the producer of single-purpose systolic arrays who can implement *any* systolic data flow in the custom network. Programmable systolic arrays can, in general, accommodate a large selection of data flows despite hardwired restrictions. This versatility implies that systolic design systems must be considered when analyzing programming languages for systolic arrays.

[e] Of course, the boards would have to be specifically designed for such scaling so that, for example, clock skew between boards does not degrade performance.

[f] The reader may wish to pursue this section directly before reading Chapter 6

Another dichotomy in the realm of systolic programming exists between those languages which abstract from hardware those which do not. Abstract programming methods are ideal for the algorithm designer who, having determined a systolic algorithm, must find a network with the appropriate data movement. Low-level programming can extract the highest performance from a programmable systolic array processor or multiprocessor. In practice, these two flavors of programming often correspond to systolic design systems and systolic programming languages, respectively. While it is clear that systolic design systems must be abstract, the converse is not required. Despite this, many systolic programming languages do not abstract from hardware, and indeed make the hardware painfully obvious to the user.

To motivate the examination of languages in Section 2.3.2, its general conclusion shall be stated forthwith. The landscape of systolic programming paradigms needs a simple language with two qualities. First, it should make use of gross hardware features without forcing the user to consider the means of data transfer. That is to say, the language should assume something about the network topology (i.e., it must be a member of a predetermined group) but not about the method of communication, be it synchronous or asynchronous. Second, it should be accessible to the programmer who desires a specific flow of data (e.g., two opposing data streams) but does not know the specific mathematical transformation necessary to arrive at that flow. Such a language would be ideal for the user of a programmable systolic co-processor. Given a board with, for example, a hexagonal mesh connection network, it should be no more difficult to program a simple linear systolic algorithm than the hexagonal version of matrix multiplication. It is hoped that the small sample of languages illustrated herein sufficiently supports these conclusions.

The abstract languages and methods have two orientations: functional and dependency. The two methods are essentially isomorphic, but both will be discussed in the next two sections.

Dependency Based Languages

Dependency mapping systems typically require some or all of the following information: a cell program, a dependency map, a network connection between processing elements, an assignment of code to individual processing elements, initialization declarations, and output declarations for the array. Some of these systems, such as Hearts[144] (nominally in this class), provide the user with graphical input methods. Although useful for the design of new systolic algorithms, it is not clear that graphical input is needed to code a known algorithm. Other work with dependency graphs includes the quasi-regular array (QRA) method,[38] HIFI,[9] SDEF,[42] the VLSI Array Compiler System (VACS),[98] and the mapping methodologies of Lee and Kedem.[102] Although it is beyond the scope of this thesis to present the methods used to derive dependency maps, their formulation is presented here, and an example of their use in Section 2.3.2.

Dependency mapping techniques have been used widely since their introduction by Moldovan.[121,123] Dependence maps are a concise method of expressing the interrelations between computation variables and loop indices in systolic algorithms expressed as nested

FOR loops. Sequential solutions to dynamic programming problems typically fall in this class. The generalized loop program is:

$$\text{FOR } \boldsymbol{i} = \boldsymbol{I}_i \text{ TO } \boldsymbol{I}_f \text{ DO } S(\boldsymbol{i}); \tag{2.12}$$

in which $\boldsymbol{i}$ is a vector of index variables bounded by $\boldsymbol{I}_i$ and $\boldsymbol{I}_f$ (or, $\boldsymbol{i} \in \mathcal{I}$ for some index set $\mathcal{I}$), and $S(\boldsymbol{i})$ is a collection of assignment statements between indexed variables, such as $X(\boldsymbol{i})$ and $Y(\boldsymbol{i})$. Sequential implementations, of course, most often increment the vector $\boldsymbol{i}$ in the fixed order of least significant index first, but most dynamic programming and other problems do not require this strictness in the order of evaluation. Dependence mapping techniques allow one to find many parallel mappings for a single problem.

A vector $\boldsymbol{d} = \boldsymbol{I}_2 - \boldsymbol{I}_1$ is a data dependency vector if the following hold:

1. $\boldsymbol{I}_1 < \boldsymbol{I}_2$ (lexicographically)

2. $Y(\boldsymbol{I}_2)$ is an output variable for some statement $s \in S(\boldsymbol{I}_2)$

3. $X(\boldsymbol{I}_1)$ is an input variable for that statement.[g]

In short, the dependency vector states that in order to do the calculation indexed by $\boldsymbol{I}$, the calculation $\boldsymbol{I} - \boldsymbol{d}$ must be done first. Multiples $\boldsymbol{d}'$ of a vector $\boldsymbol{d} \in \mathcal{D}$, the set of dependence vectors, may be ignored since if S_i depends on S_{i-2} it must also depend on S_{i-4}.

The next problem is to map the evaluation of all the index points $\boldsymbol{i}$ to space (processing elements) and time while obeying all dependency vectors. The mapping will have the following form for some embedding $\boldsymbol{E}$ (a matrix):

$$\begin{bmatrix} T \\ S \end{bmatrix} = \boldsymbol{EI}. \tag{2.13}$$

The left side of the equation is an array partitioned into a time coordinate T and one or more space coordinates S. To be a legal systolic mapping, the following must also be true:[56]

1. $\forall \boldsymbol{d} \in \mathcal{D}$, $T(\boldsymbol{Ed}) < 0$. Thus, if an operation is depended upon, it must take place before the dependent operation.

2. $\forall \boldsymbol{I} \in \mathcal{I}$, $T(\boldsymbol{EI}) \geq 0$, Thus, nothing can happen before the beginning of time.

3. $\forall \boldsymbol{I} \in \mathcal{I}$, $||P(\boldsymbol{EI})|| < \infty$. Thus, an infinite (unbounded) number of processing elements is not allowed. Negative processor indices are legal.

4. $\forall \boldsymbol{I}, \boldsymbol{I}' \in \mathcal{I}$, $P(\boldsymbol{EI}) = P(\boldsymbol{EI}') \Rightarrow T(\boldsymbol{EI}) \neq T(\boldsymbol{EI})$. Thus, two index points cannot be evaluated at the same location and the same time.

As noted, it is beyond the scope of this thesis to consider the derivation of legal mappings, but the application of this formalism to Horner's method is considered in Section 2.3.2.

[g] This is a slight modification and simplification to the work of Moldovan, intended to illustrate the main points of the mapping methodology.

Functional Languages

There are various types of functional languages. Some, such as the quasi-regular array (QRA) method[38] and DIASTOL,[56] derive systolic mappings from uniform or affine recurrence equations.[h] Alternately, Luk and Jones provide a functional language for the expression of systolic algorithms.[113] Finally, HIFI uses a graphical functional language.[9]

Since all dynamic programming problems can be expressed as uniform recurrence equations,[56] recurrence-based languages are in some sense more powerful than dependency mapping schemes. However, this difference is slight in practice, and dependence tools and techniques predominate.

Low-level Languages

Low-level languages exist on a one (or more) per machine basis. These include W2 on Warp,[21,8] Paris on the Connection Machine,[152,149] Occam on Transputer arrays,[117,65] HEP Fortran,[75,84] GAPP assembly language,[125] the ISA language,[87] and the low-level Brown Systolic Array language. Not all of these machines are primarily systolic, however the use of systolic principles can lead to fast and efficient programs on any parallel machine.[89]

2.3.2 Programming Examples

This section examines several parallel programming languages, starting with the most abstract (in the author's opinion) and ending with the most low-level. The code fragments all highlight the simple systolic algorithm for Horner's method presented in Section 2.1.2.[i] Many of the languages to be examined were not developed exclusively for programmable systolic co-processors; criticisms of these languages will be based entirely on their applicability to this field. Many of the features criticized are useful in each language's primary functions, be it the design of single-purpose systolic arrays or the programming of message-passing multiprocessors. After considering the strengths and weaknesses of these as systolic array programming language, the criteria for a new systolic language will be proposed.

SDEF

The SDEF system, developed by Engstom and Capello, requires several parts for each program specification: an index set S, domain dependencies D, a spacetime embedding E, and a process function F. The language can express any uniform systolic algorithm because of its use of dependencies and spacetime mappings. Figure 2.8 illustrates the SDEF style of programming. The SDE file gives a mathematical definition of the required

[h] Uniform recurrence equations were introduced in the seminal work of Karp, Miller, and Winograd.[78] For a discussion of the mapping of affine recurrence equations to systolic arrays, see the work of Yaacoby and Capello.[163]

[i] Most of the code fragments have not been tested as not all machines were available. Nevertheless, it is believed that the code fragments accurately display the features of each language.

```
Dimension: 2
Orthohedral Bounds:
  lower  upper
    0    degree  i
    0    n       j
Domains:
  name: C type: float dependence: i(0)  j(-1)
  name: X type: float dependence: i(-1) j(0)
  name: Y type: float dependence: i(-1) j(0)
Function: name: POLY
Embedding:
   1 1
   0 1
```

(a) SDE File

```
POLY(C,X,Y)
{
  Y' = C + X*Y;
}
```

(b) F File

Figure 2.8: SDEF program for Horner's method.

systolic data movement. There are two index variables, one for the Cs and the other for both the Xs and Ys. Each $C_{i,j}$ $(= c_i)$ depends on $C_{i,j-1}$, and is in fact equal to $C_{i,j-1}$ since the polynomial remains fixed. $Y_{i,j}$ is the partial result of the j-th x value having undergone i multiplications. SDEF uses inverted data dependency vectors, thus the vector $[0\ -1]^{\mathrm{T}}$ corresponds to $[0\ 1]^{\mathrm{T}}$ in the previous section. Referring to Equation 2.13,

$$\begin{bmatrix} T \\ S \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \boldsymbol{I}, \tag{2.14}$$

or the computation of $y_{i,j}$ takes place at time $T = i + j$ in processor $\mathcal{P}_j$.

The functional code language is a modified form of C. The programmer is relieved of worrying about how data is transferred, an advantage which will become apparent as more languages are considered. Each cell performs a transfer function (Figure 2.8b) from inputs to outputs, with the identity being the default; the output version of a variable V is named V', and if V' does not appear as an lvalue the statement $V' = V$ is assumed. Not shown in Figure 2.8 is the architectural specification which tells the SDEF system what sort of physical (or simulated) array the algorithm is to run on. The SDEF system will generate a data template describing the order in which data is to be loaded into the array in addition to code for each process or processor. SDEF is an excellent system for the development and exploration of systolic algorithms. It is, as its designers readily admit, too removed from hardware for some applications: there is an overhead associated with each process and the user cannot take advantage of low-level hardware features. A final disadvantage of the SDEF system is that, being a systolic design system, one cannot include host code and co-processor code in the same program. This means that the SDEF programmer cannot use the systolic processor as a hardware subroutine, making it difficult to link applications together. This ability is nearly a prerequisite for making full use of programmable systolic co-processors.

```
code horner_cell (c)
ports Xin, Yin, Xout, Yout;
begin real x, y, c;
   x := y := 0;
   repeat
      y := y + x * c;
      Xout <- x, Yout <- y;
      x <- Xin, y <- Yin;
   until EOS (y);
   Xout <- x, Yout <- y;
end.
```

(a) Cell program

STREAM				DESTINATION			
PAD	NAME	INDEX	DIR.	PORT NAME	DIRECTION	CODE NAME	I
1	Xvalues	1	input	Xin	east	horner_cell	1
2	ZeroFile	1	input	Yin	east	horner_cell	1
3	Xvaluesout	1	output	Xout	east	horner_cell	4
4	Yvalues	1	output	Yout	east	horner_cell	4

(b) Stream name definition

Figure 2.9: Heart's program for Horner's method.[144]

Hearts

The Hearts language, developed by Lawrence Snyder, is a less abstract systolic programming environment but is also well suited to systolic algorithm experimentation. It was developed for the Poker parallel programming system, originally created for the CHiP computers, but also used elsewhere.[144] Each Hearts program has three parts: one graphical, one in code (Figure 2.9a), and one tabular (Figure 2.9b).

The graphical part of the program is a diagram of processing elements and their connections. Through the use of a library of data flows, the user may select and propagate common graphical dependencies to all processing elements. Although the Hearts language is not itself dependency based, the library of graphical dependencies adds an important abstractness to the system. The graphical part also includes the *process assignment*, which links specific code to specific processing elements (thus defining equivalence classes), and the *port name assignment*, which assigns names to each incoming and outgoing link (such as assigning Xin and Yin to two left ports and assigning Xout and Yout to two right ports in each processing element).

The program fragment of Figure 2.9a displays several notable features.[j] The communication channels are used in a natural way; the infix operator <- can both query the ports (blocking if data is not yet available) and send data to the ports. Also, the streams

[j] For clarity, the *c* inputs are not systolically loaded into the array but passed as parameters to the cell programs. The same approach is used in the Occam example to follow.

can be queried for sentinels marking the end of the data, allowing the programmer to think asynchronously (thus, the code describes a wavefront array processor in the manner of S. Y. Kung[96]). For use as a systolic co-processor programming language, there are two aspects of Hearts which need improvement. First, the ports should be treated fully as variables in the program, allowing them to be rvalues and lvalues instead of restricting them to a single send or receive operation. Using the style of the Hearts program, the statements:

```
Yout := Yin + Xin * c;
Xout := Xin;
```

could completely express the functionality of the cell program, but are illegal. Although Hearts is reasonably abstract with respect to specific hardware, it forces too much consideration of the actual data paths. Second, the interconnection network is externally specified while the actual direction of data movement is specified in the code. The cell program should not involve itself with which way streams move, as it must since the ports have no intrinsic direction: `Xin` and `Xout` are simply port names, and a programmer could easily write a program sending data from `Xout` to `Xin`. This fragmentation of systolic movement is an unneeded complication. The simple movement of a systolic program could be better expressed with a textual statement such as

```
right stream X;
```

or some similar expression which indicates that the X values form a stream of data flowing to the right. Of course, the topology of the array must also be explicitly declared for statements like this to make sense.

The stream name assignment table (Figure 2.9b) is partly generated by the Hearts system: pad numbers and the entire destination half of the table are calculated directly from the graphical portion of the Hearts program. For each "dangling" edge of the systolic array, the user must enter a stream name, index, and direction (input or output). Stream names can be bound directly to files, simplifying data input. Also, the files may be divided up into n records of some size k. The index field of each entry refers to which of the k items should be used for that particular pad. This is especially useful for distributing matrix elements across the boundary nodes of a systolic array. Since ports have no intrinsic direction, the direction of flow must be entered in the table as well. The Hearts stream table is a reasonably good implementation of systolic array input and output. However, the inability to express a systolic program as a single unit of code is a drawback of the Hearts system.

Occam

Occam, designed by Inmos for their Transputer line of microprocessors, is an entirely textual language based on Hoare's notion of communicating sequential processes (CSP).[64] It includes asynchronous variables and parallel execution of statements. The Occam programmer is only concerned with data movement between processes, not processors. Nevertheless, systolic data movement can be forced upon the processes, as shown

```
PROC poly(VAR x[n], y[n], c[degree])=
  VAR tmp:
  CHAN Xs[degree], Ys[degree]:
  PAR
    PAR i = [0 FOR degree]
      inner (c[degree-i], Ys[i],
          Xs[i], Ys[i+1], Xs[i+1]):
    SEQ j = [0 FOR n]
      PAR
        Xs[0] ! x[j]
        Ys[0] ! 0.0
    SEQ k = [0 FOR n]
      PAR
        Xs[degree] ? tmp
        Ys[degree] ? y[k]:
```

```
PROC inner (VALUE c, CHAN Y.in, X.in,
               Y.out, X.out) =
  VAR t, x, y:
  WHILE true
    SEQ
      X.in ? x
      PAR
        t := c * x
        Y.in ? y
        X.out ! x
      y := y + t
      Y.out ! y:
```

Figure 2.10: Occam program for Horner's method.[117]

in the code of Figure 2.10. The use of streams ("channels", "ports") is similar to that of Hearts with a slight syntactic change: the `<channel> ! <var>` operation, like `<port> <- <var>`, copies a value to a shared asynchronous variable; the Occam statement `<channel> ? <var>` corresponds to `<var> <- <port>` in Hearts. The SEQ and PAR directives, with their associated indentation levels, specify statements to be executed sequentially or in parallel.

Occam has several useful features. The flow of data is expressed inside the program, though with some difficulty in the case of a systolic program: an array of channels is used to link together each "adjacent" process. The channels, as with Hearts, may be queried with infix operators. Finally, the PAR statement and associated parallel constructs can be used to avoid simple deadlocks which might otherwise occur using the channels. Unfortunately, Occam has no concept of a systolic data stream. As can be seen from the example, the programmer must go to great lengths to set up systolic behavior between the processes. With Hearts, Occam shares the use of overly present asynchronous communication. Since systolic algorithms have a regular flow of data, asynchronous variables (and the associated concern of deadlock prevention) are generally not required in systolic programming languages.

HEP Fortran

Fortran on the Denelcor HEP (Heterogeneous Element Processor) computer has two features to aid the parallel programmer.[75] First, shared asynchronous variables may be declared by prefixing a name with a dollar sign. These variables are similar to the channels of Occam or the ports of Hearts, however they have the advantage of being usable in any expression. This advantage is not without cost: the programmer must still be careful when using the shared variables since each use changes the state of the variable or port. For example the expression `J = $I*$I` might not make J a perfect

```
module polynomial (z in, c in, p out)
  float z[100], c[10], p[100];

  cellprogram (cid : 0 : 9)
  begin
    function poly
    begin
      float coeff, xin, yin, ans;
      int i;

1     receive (L, X, coeff, c[0]);
2     for i:= 1 to 9 do begin
3        receive (L, X, temp, c[i]);
4        send (R, X, temp);
      end;
5     send (R, X, 0.0);
6     for i := 0 to 99 do begin
7        receive (L, X, xin, z[i]);
8        receive (L, Y, yin, 0.0);
9        send (R, X, xin);
10       ans := coeff + yin*xin;
11       send (R, Y, ans, p[i]);
      end;
    end

    call poly;
  end
```

Figure 2.11: W2 cell program for Horner's method.

square, since the asynchronous variable $I is "emptied" each time it is accessed. The result of an expression such as $S*cos($S) will depend on the order of evaluation of the expression. Second, HEP Fortran includes the create command which spawns an asynchronous process. This is very similar to the Occam PAR statement, except that the user is restricted to executing subroutines in parallel instead of arbitrary statements.

W2

The W2 programming language is used with the Warp multiprocessor discussed in Section 2.2.3.[99] Warp's queues allow for more asynchronous behavior than the single-element shared variables of Occam and Hearts: the processors can get tremendously out of synchronization without producing the hardware blocking actions that turn off processing elements when queues become full or empty. The wavefront style of programming made possible by the queues provides for simpler initialization and result extraction than will be seen in the Paris language example to follow. An example W2 program is shown in Figure 2.11. As can be seen from the program fragment, the programmer is aware every step of the way what hardware configuration is being worked on. The send and receive statements place data on and take data from the connecting queues. Each call must specify the physical name of the queue ('X' or 'Y'), as well as which side to take the information from ('L' or 'R'). In addition, send and receive statements require variables or values, respectively, for the end processing elements to write to or read from. A more general version would allow file or stream I/O from the operating system. There is no high-level facility for specifying several data streams, so the programmer must use the hardware queues judiciously if several logical streams are desired: a slight change in the order of send and receive statements will invalidate a program. Likewise, initialization values for the queues cannot be extracted from the main body of the program

```
#define RSIZE 32   /* Allocate memory according to word size */
  #define X 0
  #define Y (X+RSIZE)
  #define C (Y+RSIZE)
  #define TMP (C+RSIZE)
  inner (xin, yout)
  real xin, *yout;
  {
   CM_f_write_to_processor_1L (0, X, xin, RSIZE);
   CM_f_write_to_processor_1L (0, Y, 0.0, RSIZE);
   CM_f_multiply_3_1L (TMP, C, X, RSIZE);
   CM_f_add_3_1L  (Y, Y, TMP, RSIZE);
   *yout = CM_f_read_from_processor_1L (degree, Y, RSIZE);
   CM_send_to_news_1L (Y, Y, 1, CM_upward, RSIZE);
   CM_send_to_news_1L (X, X, 1, CM_upward, RSIZE);
  }
  poly (x, y, c, degree, n)
  real *x, *y, *c;
  int degree, n;
  { int i;
   /* set masks, clear registers, etc */
   initialize();

   for (i=0; i <= degree; i++)
    CM_f_write_to_processor_1L (i, C, c[degree-i], RSIZE);

   /* fill it up with no output */
   for (i=0; i < degree; i++)
    inner (x[i], &y[0]);
   /* continue with output */
   for (; i < n ; i++)
    inner (x[i], &y[i-degree]);
   /* continue with no input */
   for (; i < n + degree ; i ++)
    inner (0.0, &y[i-degree]);
  }
```

Figure 2.12: Paris program for Horner's method.[149]

into a separate routine. The strong hardware orientation of W2 is too cumbersome for a general-purpose systolic language.

Paris

The Connection Machine is not, of course, a predominantly systolic processor; it is, however, a good illustration of systolic SIMD programming on a machine with many simple processing elements. The code in Figure 2.12 is a C program making use of Paris subroutine calls to control the Connection Machine. This program is worse than the W2 example: the initialization of data streams and allocation of memory is even more primitive. Since the Connection Machine is a synchronous machine, the computation must be divided into three parts, depending on how far through the array the data stream has progressed. Thus line 12 of Figure 2.12 must be repeated with slight changes to reflect each section of the computation. As with W2, there is no abstract sense of a systolic data stream; each must be implemented by telling the processing elements to send data in a given direction. Likewise, there is no abstract concept of a variable; each must be allocated by hand. Despite these disadvantages, the Paris example *does* integrate host (front end) and co-processor (the Connection Machine) functions in a single program. Thus, a single program can control file I/O and problem setup in addition to instigating co-processor computations.[k]

2.3.3 Conclusions

Having considered a selection of languages, from the mathematically elegant to the ponderous, several criteria for a systolic programming language have been developed. They are:

1. The language should cleanly separate systolic cell programs and data flow directives. Shared asynchronous variables are to be avoided because multiple references can produce unpredictable results. Instead, cell programs should be specified as pure transfer functions between systolic inputs and systolic outputs.

2. Programs should be able to execute both host and co-processor array functions, preferably in the context of a conventional language.

3. The programmer should be able to declare systolic data streams as such using concise flow directives. Stream declaration and initialization should be accomplished apart from the cell program, dissociating cell operation from macroscopic data flow and enabling the reuse of standard data flows and cell operations.

4. The language should be independent of low-level co-processor features and functions, hiding array topology, processing element architecture, array size, and the method of systolic communication from the user. The programmer should not have to specify the physical names of queues, ports, registers, or bits.

Concerning the last point, all the languages just considered have assumed the correct size systolic array, a simplification which is often not valid. Running algorithms on larger than required arrays can always be done (inefficiently) with masking, and often with more sophisticated strategies. Running on smaller than optimal arrays is more difficult, and requires algorithmic analysis. Systolic compilers should be able to perform the first partitioning and, at least, provide ways for the user to perform the second.

[k] The C* lanaguage is a somewhat higher-level programming language for the Connection machine, though as shall be seen in Appendix C it is still closely tied to the physical hardware and does not enable elegent systolic programming.

The New Systolic Language, introduced in Chapter 6, provides a framework for satisfying these requirements.

Chapter 3

Systolic Shared Register Architectures

THE Systolic Shared Register architecture is the first major conceptual contribution of this thesis. The Systolic Shared Register paradigm overcomes the cumbersome communication methods of the architectures described in Section 2.2; using its simple method of systolic communication, massively parallel VLSI co-processors may be easily constructed for any class of systolic algorithm.

3.1 Overview

This chapter presents the Systolic Shared Register (SSR) architecture. The design builds upon the problems and merits of the algorithms, architectures, and programming languages discussed previously. Perhaps the most critical issue is that of systolic communication; the simple communication of single-purpose systolic systems such as P-NAC (or the sorting array of Figure 2.1 on page 10) should be maintained, but in contrast to single- and special-purpose systolic arrays, the architecture must be fully and easily programmable. Additionally, to form very large and massively parallel co-processors, processing elements of simplicity similar to those of the Connection Machine, GAPP, or MPP are needed. However, as seen in the algorithm analysis of Section 2.1, large amounts of memory and complicated communication schemes are not required by many systolic algorithms. The four fundamental features of the Systolic Shared Register architecture are:

- broadcast instructions
- regular topology
- systolic shared registers
- a stream model of data flow.

The first two features follow directly from the definition of the class of systolic algorithms. The third feature, the use of shared registers, results from two new ways of looking at systolic co-processors. First, consider the logical structure of a traditional systolic system: an array of processing elements, with memory, connected by several nearest-neighbor data paths via communication registers or queues. More efficient systolic machines can be designed if the computational part and the storage part of the standard processing element are logically split into separate units, allowing more flexible dispositions of computation and memory throughout the VLSI chip. Second, in SSR designs there is no distinction between data memory and communication memory, somewhat reminiscent of early computer engineers discovering that the data and program memories need not be separated in hardware. Finally, the stream model of data flow, motivated by the analysis of systolic programming languages in Section 2.3, provides an elegant method for programming systolic co-processors.

3.2 Architecture Definition

In this section, each of the four defining aspects of the Systolic Shared Register architecture is considered in turn: broadcast instructions, regular topology, systolic shared registers, and data streams. A brief exploration of a generalized SSR architecture is also presented.

3.2.1 Broadcast Instructions

Per-processor programmability currently requires too much area for inclusion in high-density systolic arrays, a result of both the size of the program store and the complexity of the instruction sequencer. To form arrays with hundreds of thousands of processing elements, there should be a multitude of processing elements on each chip; otherwise, the physical space used by the co-processor would be excessively large. Typical independently programmable systolic processing elements, such as iWarp, only have one processing element per chip.[18]

Additionally, when large numbers of MIMD processors (hundreds or thousands) are used for systolic algorithms, the programs stored in each processor tend to be the similar: by definition, systolic algorithms require only a small number of cell programs.[50] Even with a relatively small number of processors, published Warp programs typically only differentiate one of the end processors while all others execute the same cell program. Similarly, excluding some relational database applications, the systolic algorithms of Table 2.1 on page 14 require at most four different cell programs: one for each equivalence class of processing elements. Although in the general case an E equivalence class program will take E times longer to execute on an SIMD machine than on an E or more instruction stream machine, some algorithms can be performed with little degradation, as noted in Section 2.1.4. Thus, multiple equivalence class programs can be efficiently executed using a single instruction stream.

The lack of exploitation of the inherent power of MIMD machines is a strong argument in favor of a single instruction stream. While a systolic instruction stream (discussed in

Section 2.2.4) is an interesting concept, broadcast instructions offer a simpler and more efficient method of control. For these reasons, SIMD broadcast instructions, which play an important part in the shared register design of an SSR architecture, are the method of choice.

3.2.2 Regular Topology

To support systolic algorithms, all of which use a regular topology by definition (see Section 1.1), the systolic co-processor itself should have a regular topology. A fixed-degree regular network with nearest-neighbor connections[a] (such as a line, square mesh, or hexagonal mesh) provides for the orderly and extensible mapping of algorithms to the systolic array. Of these possibilities, a linear network is the best choice for the major target applications of the Brown Systolic Array. A linear network extends easily and requires only two I/O streams, one for each end, no matter how large the array is. Linear systolic arrays cannot, of course, efficiently compute all of the problems mentioned in Section 2.1. Linear arrays can, however, solve a large variety of systolic problems.

This topological constraint enforces the systolic principle of locality. To each processing element, even those at the boundary, the world must look the same: there are some number of neighbors, each with a *relative* name or direction. In a mesh, these relative names might be north, south, east, and west. This is important for two reasons. First, a regular connection network is easily extensible to larger numbers of processors. Second, relative processor names greatly facilitate the use of both broadcast instructions and shared registers.

3.2.3 Shared Registers for Communication

Before turning to the inter-processor communication scheme of an SSR machine, consider the distinction between the systolic data movement of special- or single-purpose arrays and that of programmable arrays. In the special-purpose case, data movement is implicit in the hardware. For example, a processor designed specifically for a sorting problem (such as that of Figure 2.1 on page 10) might compare a value with an internally stored one and make the smaller available to the next processor all in a single step. Thus, data is pictured as moving through the array at a speed of one processing element per step. Since the processors perform only one function, the timing of the data movement can be built into the hardware.

In a programmable systolic array, data cannot be moved automatically since different applications require different types of data movement. As an extreme example, the linear systolic matrix multiplication algorithm referred to in Table 2.1 on page 14, and which will be discussed in Section 7.2, requires three different types of data movement: two streams flow in one direction at rates of one and two time steps per processor, and a third stream flows in the opposite direction at a rate of one time step per processor. Therefore, in a programmable systolic array data movement must be explicitly specified

[a] That is to say, a regular and planar network. The relaxation of this condition is considered in Section 3.2.5.

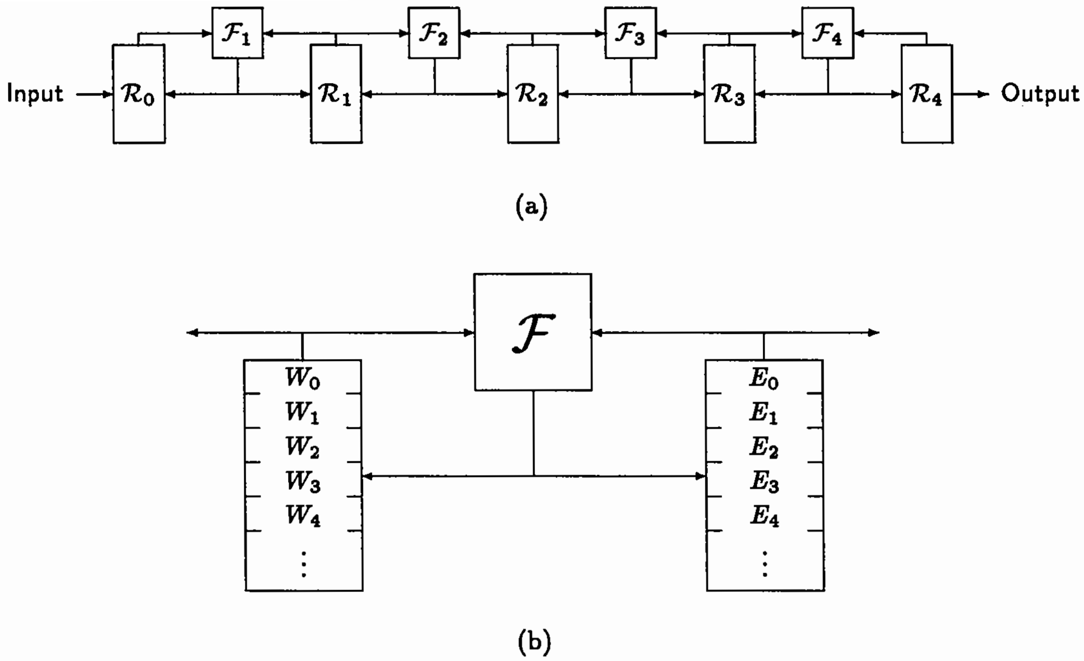


Figure 3.1: Linear SSR processor array and processor detail.

by the user. Several machines provide this ability with special instructions and special queues or registers for data movement — with Warp, this entails placing data on the X and Y queues and removing them in the correct order. In an SSR architecture, however, data moves through the array as a natural result of the user's program.

Figure 3.1 shows a segment of a linear SSR array and a detail of a single functional unit with its neighboring register banks. Note that each functional unit ($\mathcal{F}_i$) is adjacent to two register banks. Each functional unit can access data values from the register banks directly to its west and east for both input and output. Likewise, each register bank can send and receive data to and from two adjacent functional units. Input to and output from the array can take place at either end. Systolic communication in an SSR architecture uses these shared register banks. There are no internal registers in the functional units, though a small number of flag bits may be present. This separation of computation and storage is a hallmark of the SSR architecture.[b]

Every instruction broadcast to the SSR array specifies the relative addresses of memory locations: specific registers in either the west or the east register bank. During

[b] There are two vaguely similar uses of shared memory in the literature, however both are MIMD multiprocessors with standard microcomputer chips. The lattice QCD machine developed at Columbia University by Christ and Terrano features a triangular mesh of sixteen Intel 80286/80287 processor boards, each of which is able to access one of the two buses in each neighbor.[28] The other example, presented by Jagadish and others, connects Intel 8086/8087 processors with dual-ported random access memory (RAM) and an overlaying hypercube structure for message passing — again, not an example of the simple VLSI implementation of systolic communication that is the SSR paradigm.[73]

the execution of an instruction, each processing element will read operands from relatively addressed registers and, after the computation, write the result to a relatively addressed register. A single instruction may easily access two (or more in the general case) neighboring register banks. The use of broadcast instructions and relative addressing prevents contention for single-result instructions; dual-ported memory is not required since no two functional units attempt to access the same register bank simultaneously. Simple switches between functional units and register banks can control the flow of information. While there is no potential for hardware conflicts, software conflicts using the shared register banks must be avoided, as shall be discussed in Section 3.4 and again in Chapter 6. The lack of hardware conflicts is a benefit of the regular topology and broadcast instructions required by the SSR design.

With Systolic Shared Registers, data flow takes place concurrently with computation. For example, the single instruction

$$E_2 \leftarrow \min(W_0, W_1) \tag{3.1}$$

not only performs a minimization but also moves data through the array from west to east. Unlike W2 with its `send` and `receive` statements, no additional instructions are required to initiate specific types of data flow. On the bit-serial machines discussed in Section 2.2, data must be placed in communication registers before it may be accessed by adjacent processing elements. In the case of the Connection Machine, this is done with the Paris `send` instruction, in which the router accesses the processor's flag registers. The Massively Parallel Processor and GAPP machines mentioned in Section 2.2.4 have registers dedicated to communication. No extra synchronization statements or special registers are needed in an SSR machine.

The shared register method does not require the use of hardware queues between processors and does not arbitrarily limit the number of communication channels between processors; the user may decide to use the first few registers of each register bank for local storage, the next for transmission of data from west to east, the next for another data stream, and so on. In summary, shared registers provide a small, fast, flexible, and elegant implementation of systolic communication.

3.2.4 Stream Programming

Finally, SSR machines are programmed using a data stream paradigm. In, for example, the UNIX[c] operating system, utilities typically read from and write to streams (`stdin` and `stdout`) which can then be sent to other programs using the UNIX pipe operator. Thus, various utilities can filter the output stream of a program until it is in the desired form: perhaps sorted and columnized.

The data stream model for SSR machines allows the inputs and outputs at both sides of a linear array to be linked to file streams or functions by the host program. In mesh architectures, all edge processing elements, singly or in groups, may be linked to data

[c] UNIX is a registered trademark of UNIX System Laboratories in the United States and other countries.

streams. For example, a data stream of n-element vectors forming an $n \times n$ matrix could be linked to the entire west side of a square mesh. The use of a *logical* data stream is crucial: as shall be seen in Section 3.4, it is possible and, indeed, probable that a single stream of data may use different physical registers at different times, changing register meanings to produce a more efficient program. This stream model of data flow allows the programmer to ignore such detail.

Stream programming enhances the use of SSR machines as massively parallel systolic co-processors: data can be fed to and received from the array continuously and asynchronously when data queues are provided between the host and the co-processor (discussed in Section 5.2.4). The instruction of Example 3.1 uses one input and one output. The input value (perhaps the next value of a file stream) is written to the *westernmost* register bank (W_2 of $\mathcal{F}_1$ in Figure 3.1). Although this may at first seem odd, it is quite logical: $\mathcal{R}_0$ is the only register bank not written to by the minimization instruction. The output from Example 3.1 is recovered from the easternmost register bank: in addition to being written to that register bank ($\mathcal{R}_3$ in Figure 3.1), it is passed out of the array to the host. A programmable co-processor interface could discard unneeded outputs and provide default inputs without the aid of the host computer.

Unlike the other aspects of Systolic Shared Register machines, stream programming is not primarily a hardware issue: it represents a programming methodology. Stream programming will be further developed in the context of the New Systolic Language presented in Chapter 6.

3.2.5 Generalized SSR Machine[d]

The systolic shared register paradigm is not restricted to linear arrays. Figure 3.2 shows abstract designs for square, hexagonal, and octagonal mesh SSR machines. In the octagonal mesh, each functional unit can communicate with eight neighboring functional units through its four adjacent register banks.[e] Although there are two boundary edges on corner memory nodes in the hexagonal mesh and three in the octagonal mesh, just one I/O interface for each boundary register bank is needed since only one value is written to any specific register bank during the execution of an instruction. This is a direct result of the broadcast instructions and relative addressing used in SSR machines.

The SSR architecture may be further generalized as follows. A generalized Systolic Shared Register machine is an undirected bipartite graph $G = (\mathbf{F}, \mathbf{R}, \mathbf{C})$ composed of a set of function nodes $\mathbf{F}$, memory nodes $\mathbf{R}$, and communication links $\mathbf{C} \subseteq \mathbf{F} \times \mathbf{R}$. To preserve systolic features, the graph must be regular and of fixed degree. These two qualities ensure that graphs of arbitrary size may be constructed, a requirement for massively parallel systolic systems. For VLSI systems, G will be planar or toroidal.

[d] Although this formal definition of the SSR architecture is not necessary for understanding B-SYS, it is important to precisely define what has been only intuitively described.

[e] Note that the hexagonal and octagonal meshes featured in Figure 3.2 have the same number of function and memory nodes (like the linear array), while the square mesh has twice as many memory nodes. A hexagonal mesh could, of course, be implemented with a degree 2 network which would use three times as many register banks as functional units.

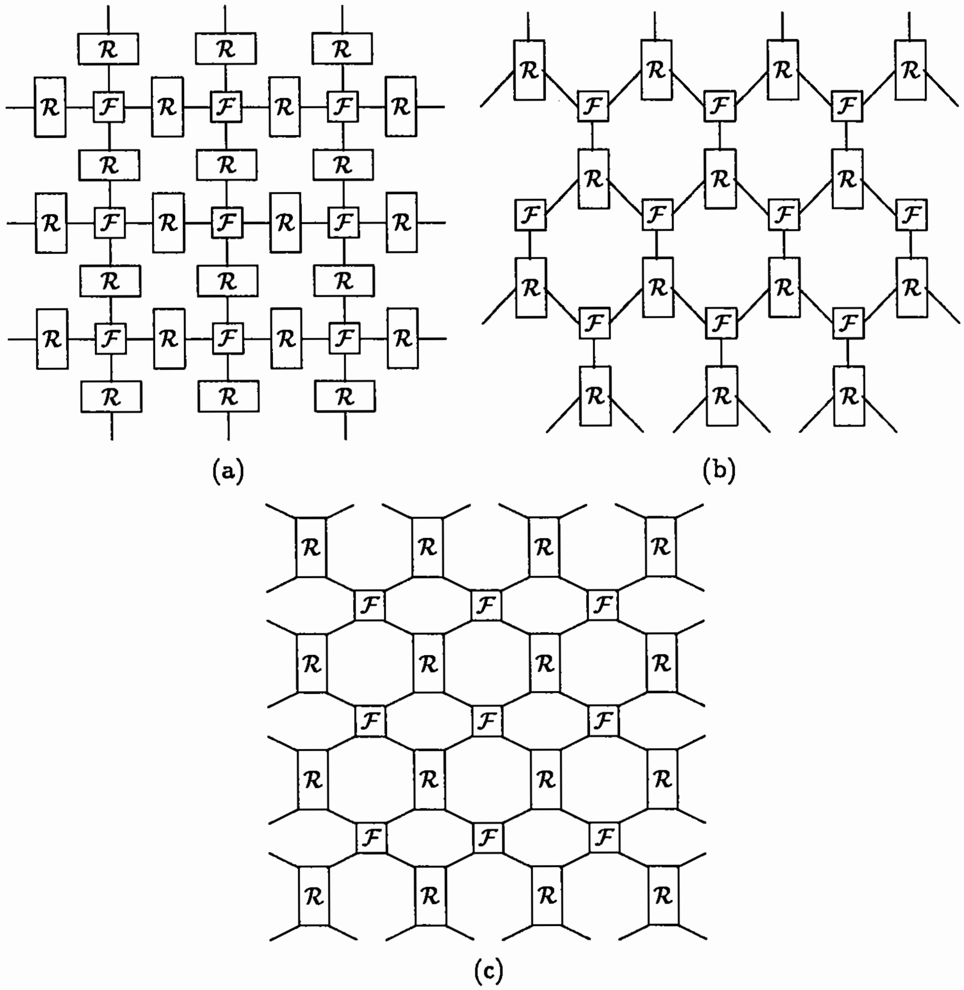


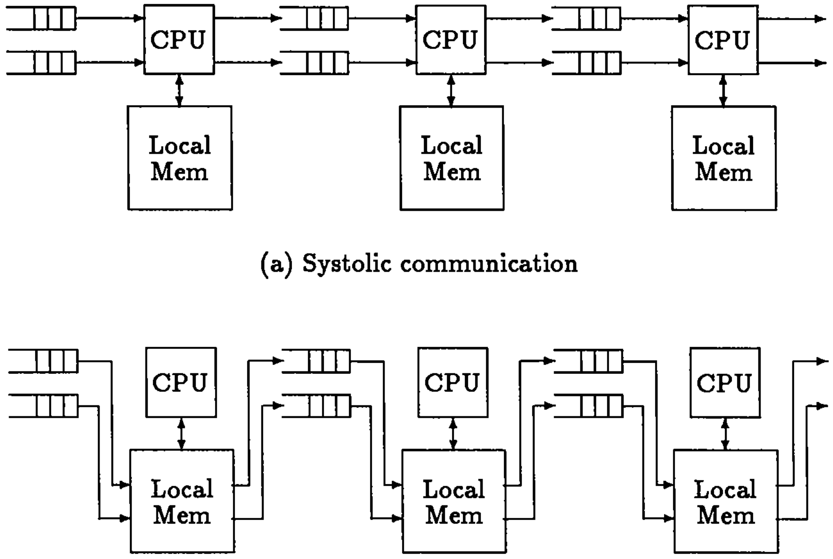
Figure 3.2: SSR networks: (a) square, (b) hexagonal, and (c) octagonal mesh.

Cube-connected cycle networks (degree 3) may also use this paradigm. If the fixed degree restraint is relaxed, hypercube ($|\mathbf{C}| = n \log n$) and shared memory ($|\mathbf{C}| = n^2$) extensions to the SSR concept result. In general, $\mathcal{R}_{i,j}$ refers to the jth element of register bank i, while $\mathcal{F}_i$ is the ith functional unit.

Function nodes $\mathcal{F}_1, \mathcal{F}_2 \in \mathbf{F}$ *share* memory node $\mathcal{R} \in \mathbf{R}$ if $(\mathcal{F}_1, \mathcal{R}) \in \mathbf{C}$ and $(\mathcal{F}_2, \mathcal{R}) \in \mathbf{C}$ or, alternatively, if $\text{dist}(\mathcal{F}_1, \mathcal{F}_2) = 2$. A functional unit $\mathcal{F}$ has access to all registers in its neighborhood

$$\text{nghd}(\mathcal{F}) = \{\mathcal{R}_{i,j} \in \mathcal{R}_i : \mathcal{R}_i \in \mathbf{R} \text{ and } (\mathcal{R}_i, \mathcal{F}) \in \mathbf{C}\}.$$

As with the linear case, register banks are referred to relative to a functional unit (east, northwest, etc.): the regularity of the network and the SIMD control assure that multiple accesses of the same memory bank will not occur.



(a) Systolic communication

(b) Memory-to-memory communication

Figure 3.3: Methods of inter-processor communication.[90]

Each instruction in a program $I^1, I^2, \ldots, I^t, \ldots$ performs a computation on some set of neighboring registers to determine the values of some other group of neighboring registers, thus each is of the form:

$$R_1^t \leftarrow f(R_2^{t-1}, R_3^{t-1}) \qquad R_1, R_2, R_3 \in \text{nghd}(\mathcal{F}).$$

R_1^t refers to register location 1 at time t The instruction can easily be generalized to different numbers of operands and results (for example, a flag operand and result are used in B-SYS). Each instruction I^t is executed simultaneously in all functional units at time t. The presence of a mask or context flag, such as that in B-SYS, corresponds to including the mask flag and the original value of the result register(s) as operands to each instruction.

The Systolic Shared Register architecture is not just the foundation of the Brown Systolic Array; it is a versatile framework for parallel computing. Although best suited for VLSI, in which register banks and functional units may be arbitrarily positioned with ease, its structure may be used for any parallel machine.

3.3 An Evaluation of SSR Communication

It is instructive to examine the Systolic Shared Register design in light of the definition of systolic communication proposed by H. T. Kung.[90,91] Kung separates memory-to-memory communication and systolic communication. In the former, information is passed from the memory of one processing element to the memory of another via communication queues. In systolic communication, however, information may be both directly read from and written to the communication queues by the central processing unit. In both cases, there is a distinction between the queue memory used for communication and each processor's own local memory. Figure 3.3 shows these two methods for providing communication between processing elements in a linear array.

This distinction between the two types of memory is important: the communication memory defined by Kung only allows sequential access (as with Warp), while the local memory allows random access. Thus, a systolic array machine is one that can make direct use of data on its input queues. Since the register banks of an SSR architecture are essentially a set of single-element, bi-directional queues, the SSR architecture is systolic by this definition. However, the SSR method has taken the idea of systolic communication one step farther: the shared register banks allow random access, making local memories superfluous. Since SSR systems do not distinguish between memory types, the CPUs not only communicate with their communication registers but compute with them as well.

Another difference between communication queues and shared registers is that the SSR paradigm's use of communication registers forces synchronous execution on the system. If an item is to be read from a specific register, it must be written to that register by an adjacent processor before that time. This is a simple task with synchronous broadcast instructions and closely resembles the data movement present in special-purpose systolic arrays. On the whole, Systolic Shared Registers provide more efficient systolic communication than systolic queues.

3.4 Programming Techniques

Moving data through an SSR machine is more complicated than simply reading two operands from the west register bank (W) and storing the result in the east register bank (E). Since E is the west register bank of an adjacent functional unit $\mathcal{F}$, writing to it may invalidate data used by $\mathcal{F}$. Taking another look at the sorting algorithm of Figure 2.1, the operation of storing the maximum and passing the minimum has been conveniently lumped into one instruction. This operation may be represented as:

$$E_1, W_0 \quad \leftarrow \quad \text{minmax}(W_0, W_1), \tag{3.2}$$

where register 0 is used for local storage of the maximum and register 1 is used for communication. Since instructions typically have only one output register, this version is unrealistic. A possible translation is:

$$\begin{array}{rcl} E_1 & \leftarrow & \min(W_0, W_1) \\ W_0 & \leftarrow & \max(W_0, W_1). \end{array} \tag{3.3}$$

This program appears to have the same effect as Example 3.2, however E_1 for the i-th functional unit $\mathcal{F}_i$ is W_1 of $\mathcal{F}_{i+1}$; the writing to E_1 by $\mathcal{F}_i$ will invalidate the maximum computation of $\mathcal{F}_{i+1}$, and so on down the line. There are two solutions to this problem: the use of scratch registers and the use of phased programs.

3.4.1 Scratch Registers

Scratch registers are the obvious solution to the problem present in Example 3.3. Using W_2 as a scratch register, the program becomes:

$$\begin{array}{rcl} W_2 & \leftarrow & \min(W_0, W_1) \\ W_0 & \leftarrow & \max(W_0, W_1) \\ E_1 & \leftarrow & W_2. \end{array} \tag{3.4}$$

At the cost of one register and one instruction, the overwriting problem has been solved. For this simple sorting algorithm, the extra instruction causes a large degradation in performance. A phased systolic program will provide a more efficient solution.

3.4.2 Phased SSR Programs

The second alternative is the use of a *phased* program. A phased program has several distinct phases during which different registers are used for communication. For example, the communication register might be E_2 in phase 1 and E_1 in phase 2. Switching back and forth between these two phases (called *weaving*) will result in a correct program which, though an additional register is still used, does not require any extra instructions. Two inner loops of the sorting program would then be:

$$\begin{array}{rcl} E_2 & \leftarrow & \min(W_0, W_1) \\ W_0 & \leftarrow & \max(W_0, W_1) \\ E_1 & \leftarrow & \min(W_0, W_2) \\ W_0 & \leftarrow & \max(W_0, W_2). \end{array} \tag{3.5}$$

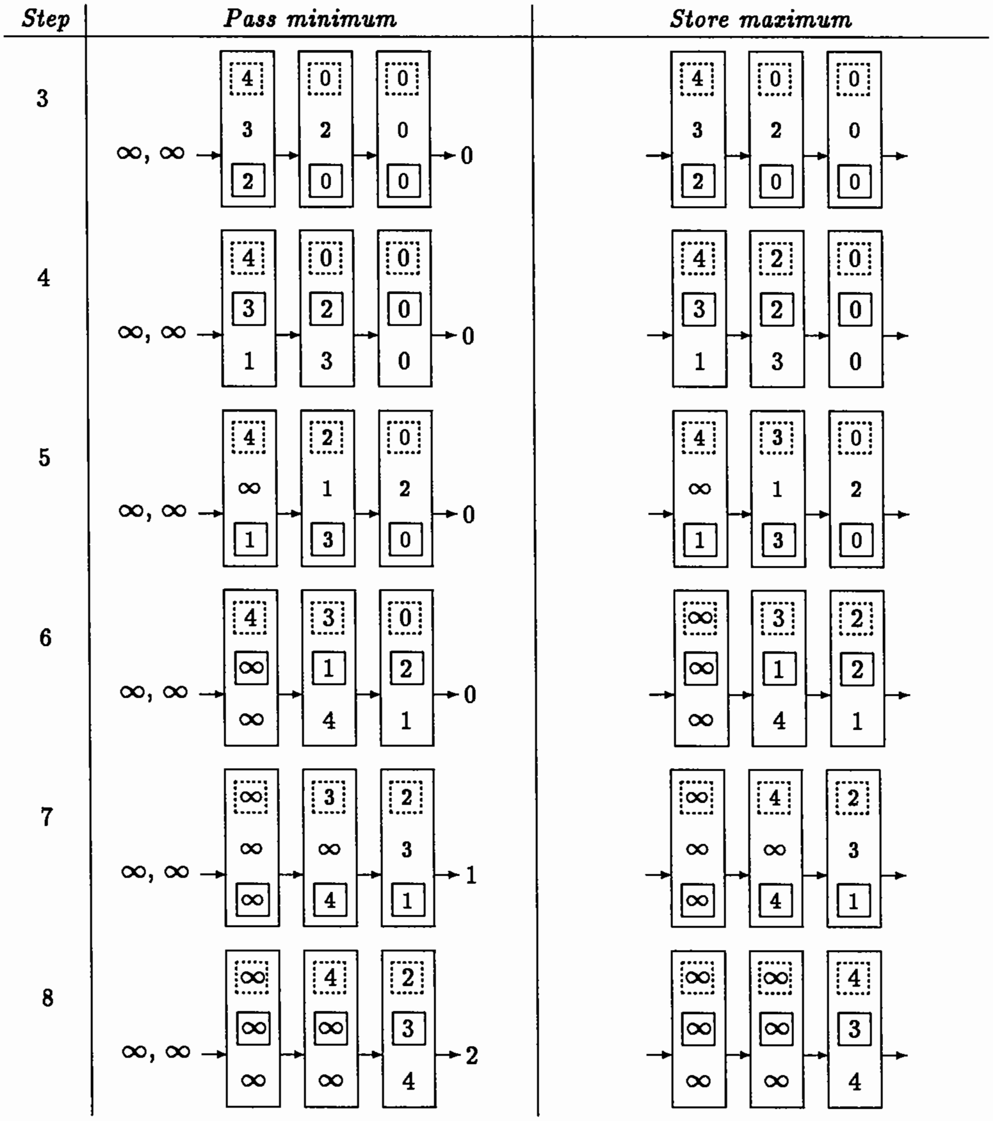
Sorting with this technique is shown in Figure 3.4. The current scratch register is boxed, while the top register is used for local storage. During the first part of each phase, the minimum of the boxed scratch value and the top stored value is passed to the unused register of the adjacent processing element. During the second part, the maximum of the two values is locally stored.

This technique can also be applied to programs with more than two phases. For example, suppose a half-speed data stream is flowing from west to east. The general three-phase program would be:

$$\begin{array}{rcl} E_0 & \leftarrow & f(W_2) \\ E_2 & \leftarrow & f(W_1) \\ E_1 & \leftarrow & f(W_0). \end{array} \tag{3.6}$$

This data stream may then be combined with a two-phase data stream to form a program with six (the least common multiple of two and three) phases.

The use of phased programs to avoid software conflicts will be further considered in Chapter 6, wherein the New Systolic Language's automatic support for phased programs will simplify the programming of Systolic Shared Register machines.



The local storage register is enclosed in a dotted box. The scratch register is shown in a solid box. The remaining register is used to receive data from the adjacent processing element. Functional units are not shown.

Figure 3.4: Phased systolic sort.

3.5 Conclusions

This chapter has presented a new architecture for massively parallel systolic processing: the Systolic Shared Register architecture. The SSR architecture combines the best features of special-purpose systolic arrays (many processors, synchronous data movement) with the power of programmable arrays. The four attributes of the Systolic Shared Register architecture are:

- SIMD broadcast instructions for control
- regular interconnections for systolic data flow
- shared registers for communication and computation
- data streams for programming.

The SSR paradigm is applicable to all types of programmable systolic co-processors, including linear arrays and various types of meshes. Although SSR programming has some intricacies, Systolic Shared Registers provide a simple and practical implementation of systolic communication.

Chapter 4

Design of the Brown Systolic Array

THE basic architecture of the Brown Systolic Array is that of a linear Systolic Shared Register machine. Apart from this, there are many choices left open in the generic architecture which must be considered for each SSR implementation: functional unit complexity, register bank size, and word size are the most obvious. Proceeding down to the lowest design level, even after these architectural issues have been decided, potential implementations of the chosen features must be suitably analyzed to produce a working chip. This chapter considers all aspects of the B-SYS design, from the architecture down to the CMOS layout.

4.1 Architecture

Recall from Section 2.2 the three types of parallel machine examined. The single-purpose systolic array could achieve massive parallelism because of its size but was not programmable. The SIMD bit-serial machine was also massively parallel but had more memory and routing capabilities than is required for combinatorial problems. The elimination of these features could provide either the same processing power (for combinatorial problems) in a smaller space or a much more massively parallel machine in the same space. Space, of course, corresponds directly to cost. The use of a bit-parallel processing unit could improve performance as long as pin requirements remain feasible. Finally, the MIMD machine lacked the ability to become massively parallel in a reasonable amount of physical space: the processing elements were simply too large. The Brown Systolic Array, being an implementation of the SSR architecture, solves many of these problems for the domain of combinatorial problems.

Developing further the generic linear SSR array of Figure 3.1 on page 40, Figure 4.1 illustrates a B-SYS functional unit and its two adjacent register banks. The Brown Systolic Array uses an 8-bit word, large compared to bit-serial processor arrays and small compared to MIMD machines. For combinatorial problems, with their reliance on

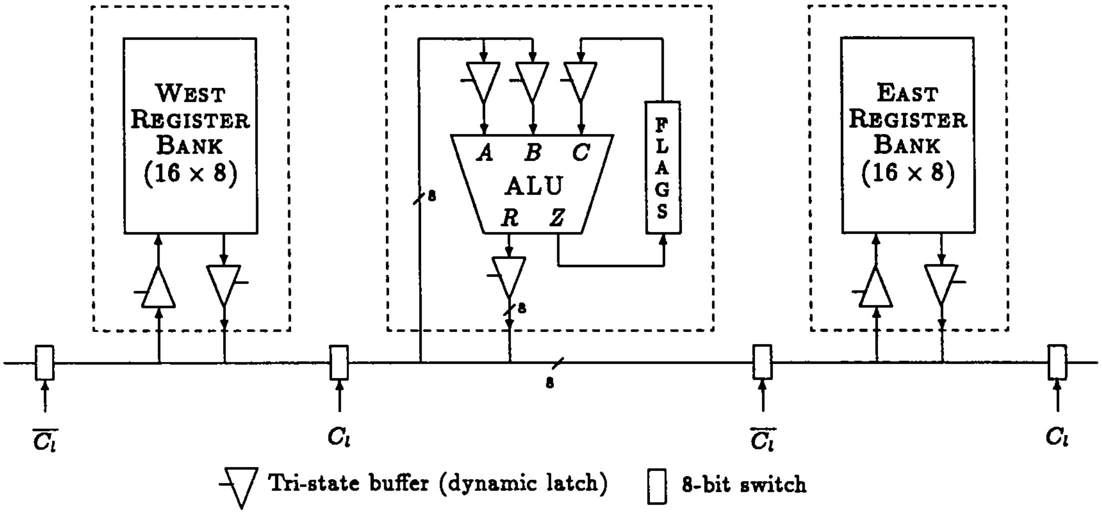

West Register Bank (16 x 8)
A B C
ALU
R Z
F L A G S
8
8
8
East Register Bank (16 x 8)
$\overline{C_l}$
C_l
$\overline{C_l}$
C_l
Tri-state buffer (dynamic latch)
8-bit switch

Figure 4.1: Segment of the B-SYS array.

symbols and small numbers, 8-bit words are sufficient. Common ASCII[a] text uses seven or eight bits, thus B-SYS is well suited for text compression, comparison, and other applications. Although an 8-bit ALU cannot be expected to perform eight times faster than a 1-bit ALU, it is in general faster.

Each register bank has sixteen 8-bit registers, the same amount of storage per processing element as the GAPP chip (Section 2.2.4), and an amount suitable for many sequence comparison and other symbolic computations. Access to the register banks is controlled by complementary 8-bit switches on each side of the register bank.

Functional units have eight local flags ($F_0 \ldots F_7$) which are used for the storage of ALU flag inputs and outputs, as well as the processing of conditionals. One of the flags (F_0) is a context flag which enables the conditional execution of instructions. If the command broadcast to the array enables use of the context flag, only those with a set flag will write the ALU results to memory.[b]

4.1.1 B-SYS Arithmetic Logic Unit

The Brown Systolic Array's arithmetic logic unit, shown in Figure 4.2a, has two one-word inputs, A and B, and a single-bit input C. The outputs are a result word R and a bit Z. The ALU has a bit-slice architecture: each bit of the output is computed in a manner identical to all other bits in the ALU and, indeed, all other bits on the chip.[c]

[a] American Standard Code for Information Interchange, ANSI standard X3.4-1977.

[b] Actually, a result is written to a register bank $\mathcal{R}_i$ if the functional unit to its east ($\mathcal{F}_i$) has an asserted context flag. Thus, when the destination is in the east register bank, the writing of a result R will not be controlled by the functional unit $\mathcal{F}_i$ which generated it, but by $\mathcal{F}_{i+1}$. The writing of $\mathcal{F}_i$'s flag output is always controlled by $\mathcal{F}_i$'s context flag. This is caused by an early design flaw which, when discovered, was deemed too costly to fix.

[c] A note on fonts used in the text. Architectural names are expressed as CP, p_i, and the like. Low-level control signals and opcodes are expressed as K1, xorAB, and the like. Hexadecimal numbers are

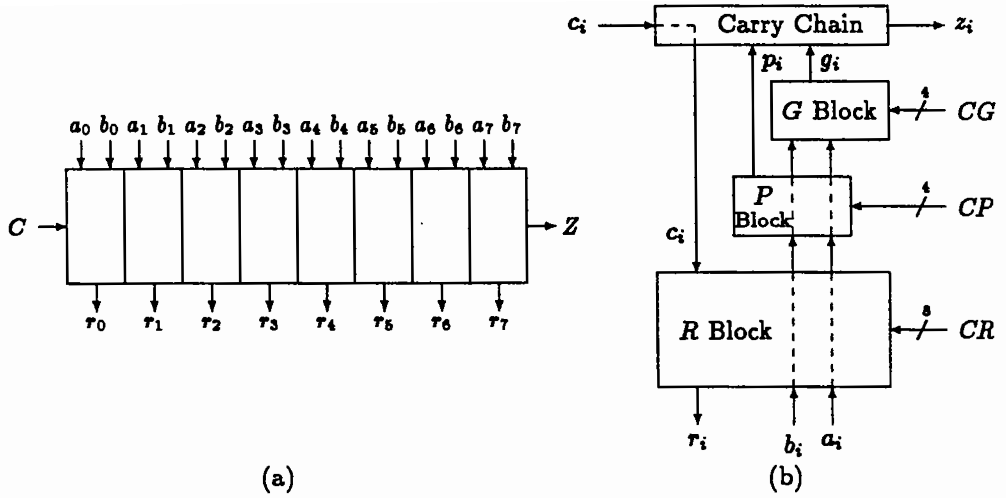

a_0 b_0 a_1 b_1 a_2 b_2 a_3 b_3 a_4 b_4 a_5 b_5 a_6 b_6 a_7 b_7
C
Z
r_0 r_1 r_2 r_3 r_4 r_5 r_6 r_7
(a)
c_i
Carry Chain
z_i
p_i
g_i
G Block
4
CG
P Block
4
CP
c_i
R Block
8
CR
r_i
b_i
a_i
(b)

Figure 4.2: B-SYS arithmetic logic unit and bit-slice detail.

c_i	b_i	a_i	g_i	p_i	r_i	z_i
			$a_i \cdot b_i$	$a_i \oplus b_i$	$a_i \oplus b_i \oplus c_i$	$g_i + p_i \cdot c_i$
0	0	0	0	0	0	0
0	0	1	0	1	1	0
0	1	0	0	1	1	0
0	1	1	1	0	0	1
1	0	0	0	0	1	0
1	0	1	0	1	0	1
1	1	0	0	1	0	1
1	1	1	1	0	1	1

Table 4.1: Carry lookahead addition.

The i-th bit-slice, detailed in Figure 4.2b, is loosely based on the design of the OM-1 arithmetic logic unit from Mead and Conway.[120] A carry lookahead ALU is used: generate and propagate signals are calculated from the a_i and b_i inputs and then used to calculate the carry bit output z_i. The result output r_i is computed from the a_i and b_i inputs as well as the carry input c_i (equal to z_{i-1} from the previous stage). The generate and propagate signals have their standard meanings: if g_i is asserted then z_i is asserted. Otherwise, if p_i is asserted then $z_i = c_i$. If neither of these cases hold, z_i is not asserted.

Arbitrary function logic blocks compute p_i, g_i, and r_i. In the former cases, any 2:1 function (specified by CP and CG) and in the latter any 3:1 function (specified by CR) may be used. For example, consider the addition operation. In this case, a carry is generated at the i-th stage if a_i and b_i are both 1. A carry is propagated though the i-th stage if either of the inputs is 1. The sum bit is the exclusive-or of the inputs ($a_i \oplus b_i \oplus c_i$).

expressed as $6C_{16}$ or 6C.

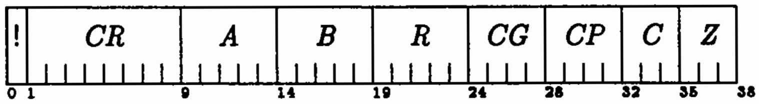


Figure 4.3: B-SYS instruction word.

The logic block control word is the appropriate column of the function table. In the case of addition, the result function $CR = 96_{16} = 10010110_2$ can be read from the r_i column of Table 4.1 (the most significant bit of CR corresponds to $a_i = b_i = c_i = 1$ in the table). The generate and propagate control signals, independent of c_i, are four bits long. Reading from the top half of the table, the generate function is $CG = 8_{16}$ and the propagate function is $CP = 6_{16}$. The propagate function $CP = \mathrm{E}_{16}$ would be equally valid for addition, corresponding to $p_i = a_i + b_i$. As can be seen from the above table, the order of significance for inputs to the function blocks is c, b, and then a.

B-SYS Microcode

The 38-bit horizontal microcode format of a B-SYS instruction is shown in Figure 4.3. The instruction word may be conveniently divided into two major parts: the register input and output addresses, along with CR, and the flag input and output addresses, along with CG and CP. Each broadcast instruction has the following fields:

- !: When asserted, all functional units execute the broadcast instruction regardless of the context flag. Otherwise, only those functional units with a 1 in flag F_0 will store the ALU results. This field corresponds to the Call control line of the B-SYS chip.

- CR: eight bits expressing the 3:1 function used to calculate R.

- A: one bit specifying a register bank (left or right) and four bits specifying which register in that bank is the A input to the ALU.

- B: five bits, as with A.

- R: five bits, as with A, specifying the register in which to store the result.

- CG: four bits expressing the 2:1 function for the generate signal of the carry chain.

- CP: four bits expressing the 2:1 function for the propagate signal of the carry chain.

- C: three bits specifying the address of the input flag to the ALU (c_0).

- Z: three bits specifying the address for the output flag from the ALU (z_7).

All fields are required and there are no immediate operands. The CR, CG, and CP control words are broadcast to each bit-slice of the ALU, while the remaining fields

Name	Function	CR_{16}	Name	Function	CR_{16}
zero	0	00	one	1	FF
fnA	a	AA	fnC	c	F0
xorAC	$a \oplus c$	5A	andAC	ac	C0
notAxorC	$\bar{a} \oplus c$	A5	xorAB	$a \oplus b$	66
xnorAB	$a \odot b$	99	andAB	ab	88
nandAB	$\overline{a \cdot b}$	77	orAB	$a + b$	EE
norAB	$\overline{a + b}$	11	xorABC	$a \oplus b \oplus c$	96
selectABonC	$ac + b\bar{c}$	AC	selectABonC_	$a\bar{c} + bc$	CA

Table 4.2: B-SYS result block control words.

Name	Function	CG_{16}	CP_{16}	C
Zzero	0	0	0	ϕ
Zone	1	F	0	ϕ
Zadd	carry	8	6	0
Zadda	increment A	0	A	1
Zsub	borrow	4	9	0
Zconst	C	0	F	ϕ
notzeroA	$a \neq 0?$	A	F	0
zeroA	$a = 0?$	0	5	1
matchAB	$\exists i : a_i = b_i?$	8	F	1
nomatchAB	$\forall i, a_i \neq b_i?$	0	6	1
equalAB	$A = B?$	0	9	1
notequalAB	$A \neq B?$	6	F	0

Table 4.3: B-SYS generate and propagate control words.

are used by the processor to access the correct memory elements. If the user only needs one of the two results (R and Z), the other must be placed in a scratch register or flag. If a program requires certain constants, they must be loaded into one of the registers beforehand. Often, the first step of a program will be to place values in several flag registers and clear several locations in the register banks. Constants needed in all processing elements can be constructed in a small number of B-SYS statements without having to shift them through the array.[d]

B-ASM Assembly Language

Table 4.2 displays the hexadecimal codes for several result logic block control words. The selection command selectABonC chooses between operands according to a flag input; it is vital for the processing of conditionals without the use of the mask flag. Since masking operations require additional instructions to set and clear F_0, register selection is often the most efficient way to process conditionals.

[d] For example, registers can be set to arbitrary values in eight steps by shifting in ones and zeros from the flags.

```
#include "../src/opc2.h"

/* Loop program for a phased systolic sort.                 */

/* W0 stores the maximum value encountered so far.          */
/* W1 and W2 alternate as communication registers.          */
/* W3 is a scratch register for the unneeded difference.    */
/* F1 stores comparison results                             */
/* F2 is clear                                              */
/* The mask flag (F0) is not used.                          */

/* read:  host reads a result from the co-processor.        */
/* write: host writes an input to the co-processor.         */


! xorABC       W2 W0 W3 nread nwrite Zsub   F2 F1    /* W2 - W0            */
! selectABonC W2 W0 E1  read  write Zconst F1 F1    /* E1 = min (W2, W0) */
! selectABonC W0 W2 W0 nread nwrite Zconst F1 F1    /* W0 = max (W2, W0) */

! xorABC       W1 W0 W3 nread nwrite Zsub   F2 F1    /* W1 - W0            */
! selectABonC W1 W0 E2  read  write Zconst F1 F1    /* E2 = min (W1, W0) */
! selectABonC W0 W1 W0 nread nwrite Zconst F1 F1    /* W0 = max (W1, W0) */
```

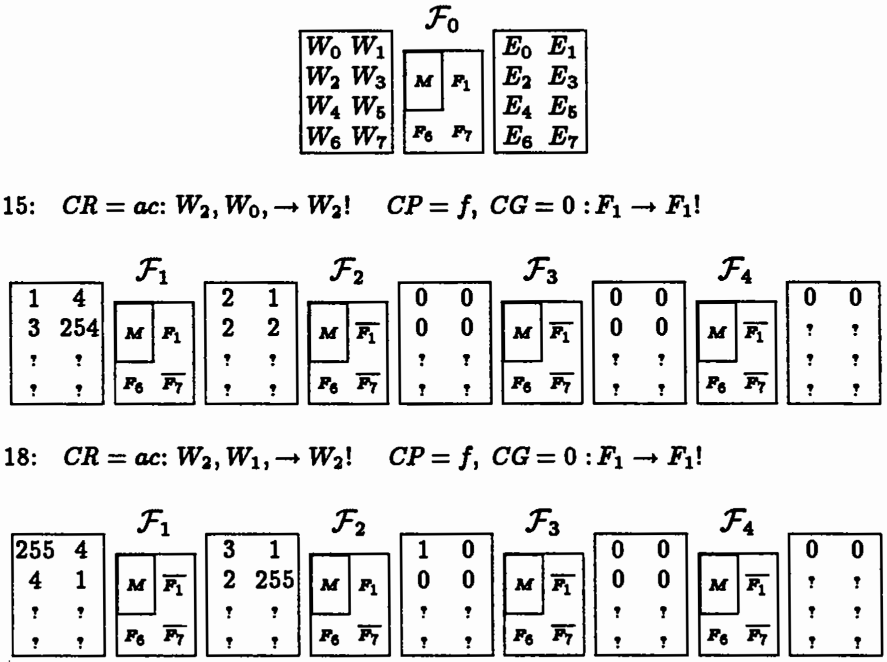
Figure 4.4: B-SYS phased systolic sort.

On appropriate flag inputs the Manchester carry chain can perform several test functions. For example, with $C = 1$, $CP = 5_{16}$, and $CG = 0_{16}$, a carry will never be generated, and the input carry will be propagated only if $a_i = 0$. Thus, these two codes test the A operand for equality to zero. Several generate and propagate control words are shown in Table 4.3.

Figure 4.4 shows a program for the phased systolic sorting algorithm of Figure 3.4 on page 47. The program (from the Brown Systolic Array Simulator) uses the C preprocessor as an assembler. In addition to the instruction fields just described, each command also has information describing host I/O: when results should be written to and read from the co-processor board, as described in Section 3.2.4. Programs such as this are used both when programming the Brown Systolic Array and when using its simulator.

4.1.2 Architecture Simulation

The Brown Systolic Array Simulator (B-SIM) proved crucial for algorithm development and early evaluation of the B-SYS architecture. The simulator emulates the data flow between the host and the systolic co-processor using UNIX pipes: when running the simulator, the host process analyzes the input assembly code with its primitive looping construct, combines instructions with input data, and passes the information to the board process. The board process simulates the array and, when requested, passes information back to the host process which can then send data to the appropriate files. The host



$\mathcal{F}_0$
W_0 W_1 W_2 W_3 W_4 W_5 W_6 W_7
M F_1 F_6 F_7
E_0 E_1 E_2 E_3 E_4 E_5 E_6 E_7
$\mathcal{F}_1$ $\mathcal{F}_2$ $\mathcal{F}_3$ $\mathcal{F}_4$
1 4 3 254 ? ? ? ?
M F_1 F_6 $\overline{F_7}$
2 1 2 2 ? ? ? ?
M $\overline{F_1}$ F_6 $\overline{F_7}$
0 0 0 0 ? ? ? ?
M $\overline{F_1}$ F_6 $\overline{F_7}$
0 0 0 0 ? ? ? ?
M $\overline{F_1}$ F_6 $\overline{F_7}$
0 0 ? ? ? ? ? ?
$\mathcal{F}_1$ $\mathcal{F}_2$ $\mathcal{F}_3$ $\mathcal{F}_4$
255 4 4 1 ? ? ? ?
M $\overline{F_1}$ F_6 $\overline{F_7}$
3 1 2 255 ? ? ? ?
M F_1 F_6 $\overline{F_7}$
1 0 0 0 ? ? ? ?
M $\overline{F_1}$ F_6 $\overline{F_7}$
0 0 0 0 ? ? ? ?
M $\overline{F_1}$ F_6 $\overline{F_7}$
0 0 ? ? ? ? ? ?

Time 15: $CR = ac$: $W_2, W_0, \rightarrow W_2!$ $CP = f$, $CG = 0 : F_1 \rightarrow F_1!$

Time 18: $CR = ac$: $W_2, W_1, \rightarrow W_2!$ $CP = f$, $CG = 0 : F_1 \rightarrow F_1!$

Figure 4.5: B-SIM TEX snapshot.

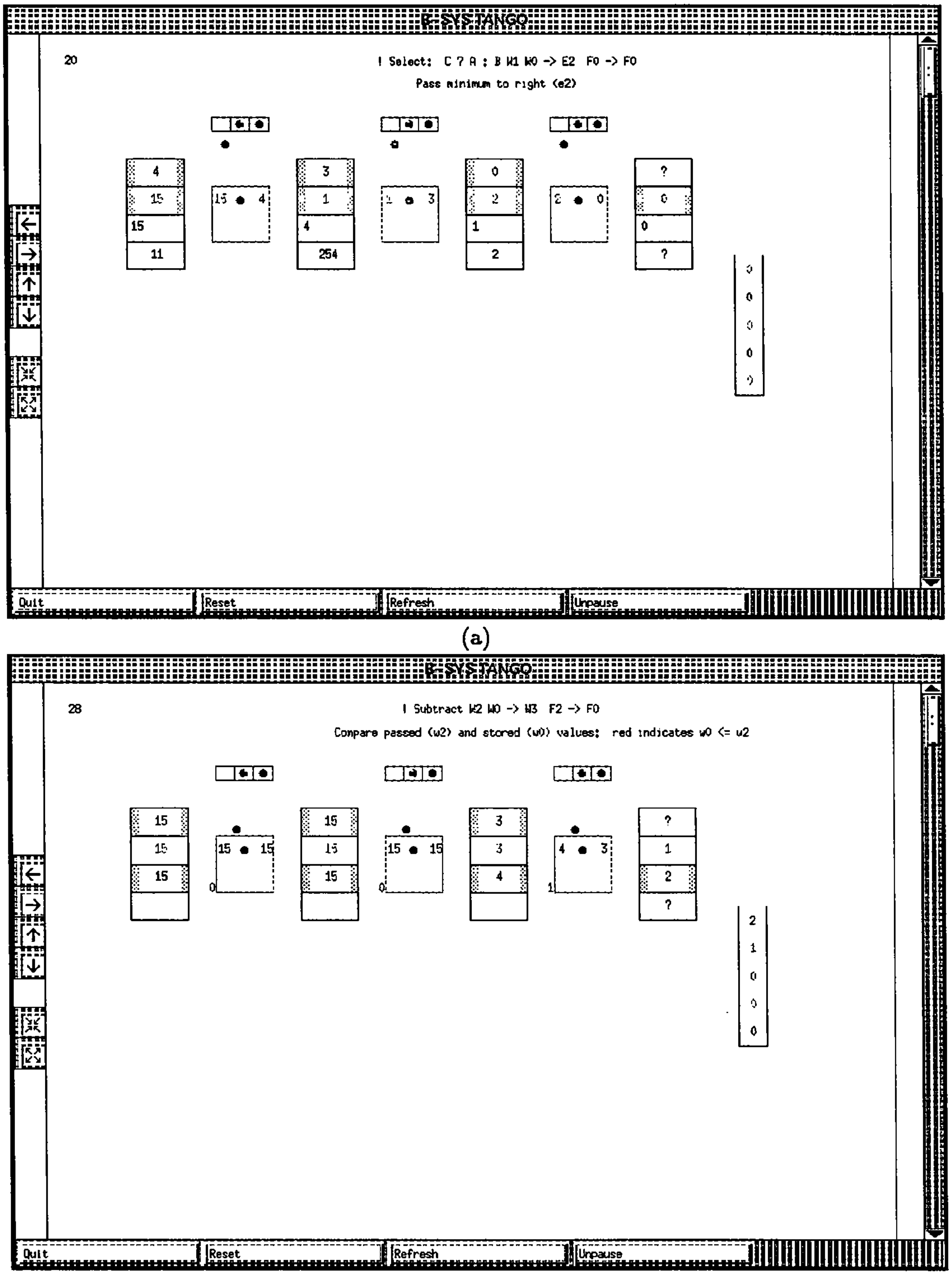
process can be modified for different programming models without change to the basic simulator.

In addition to passing information back to the host process, the board process can (on command) write the state of the entire array to a file. Using another program, this raw data from the board process can be transformed into TEX snapshots of the array similar to that of Figure 4.5. As an added feature, the board process can also send FIELD messages to an independently running systolic array animation system based on the Tango package.[134,145] With the animator, the user can see operands move to the functional units and results move back to the appropriate register bank. Flags are represented with red (clear) and green (set) circles. Although, given a B-SIM program, the animation is entirely automatic, the user may also elect to include comments and pause commands at specific points in the animation and to highlight certain registers in the array. Such highlighting can reflect register bank meanings in a phased systolic sort, as illustrated by Figure 4.6. (Of course, two black and white frames do not do justice to the color Tango animation.)

The use of the Brown Systolic Array simulator is detailed in Appendix A.

4.2 Design

This section tours the design of the Brown Systolic Array. After a review of CMOS technology, the overall organization of B-SYS is described. As shall be seen, B-SYS



20

! Select: C ? A : B W1 W0 -> E2 F0 -> F0

Pass minimum to right (e2)

(a)

28

! Subtract W2 W0 -> W3 F2 -> F0

Compare passed (w2) and stored (w0) values; red indicates w0 <= w2

(b)

Figure 4.6: B-SIM Tango animation.

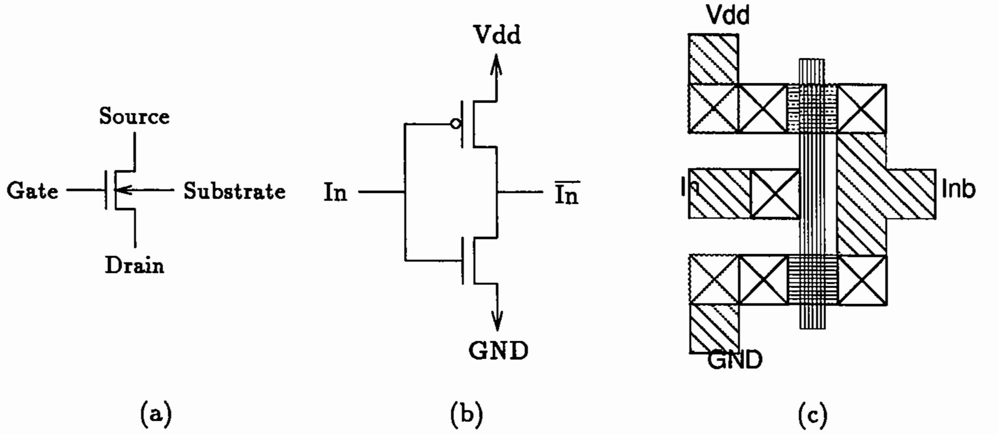


Figure 4.7: Example CMOS circuits: transistor, inverter, and inverter layout.

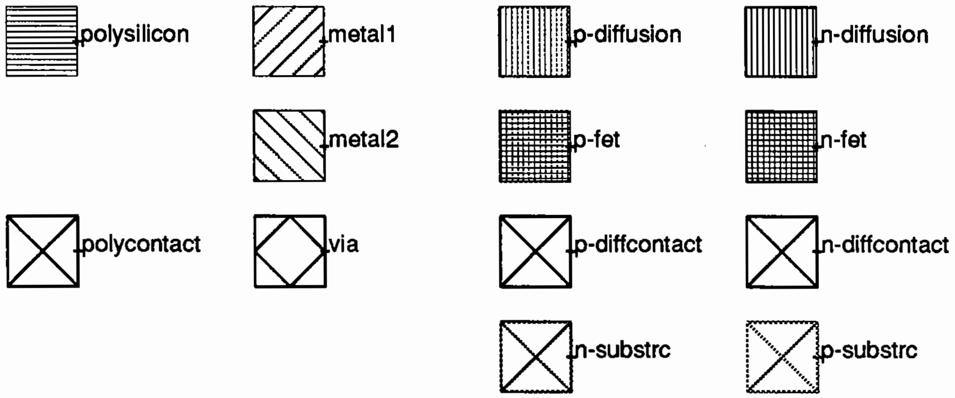
packs 85 000 transistors on a 6.9 mm × 6.8 mm chip, a density made possible by the simplicity and elegance of the SSR design. Finally, the implementation of various B-SYS subsystems is considered.

4.2.1 CMOS VLSI

Before proceeding, a brief review of (or introduction to) digital Complementary Metal Oxide Semiconductor (CMOS) VLSI is in order. As the name implies, CMOS has two basic circuit elements: the nFET (n-type field effect transistor) and the pFET (its complement). Digital CMOS circuits are formed with these two transistors, used as switches, and the supply voltages power (Vdd) and ground (GND). In the ideal circuit, there are no analog effects; that is to say, all signals match (in voltage) one of the supply voltages and all switches both do and do not conduct perfectly, depending on state. CMOS comes very close to this ideal case, certainly much closer than such technologies as nMOS (CMOS without the pFET) and transistor-transistor logic (TTL).

The n-transistor of Figure 4.7a is a positive switch: when its gate is connected to a high voltage (Vdd) the n-transistor will turn on, connecting the source and the drain terminals. The complementary p-transistor is a negative switch: when its gate is connected to a low voltage (GND) the p-transistor will conduct. CMOS transistors are nearly perfect switches. No current is required to maintain the FET's conductive state, although some current is required to *change* a transistor's state because of the gate capacitance. Also, nFETs conduct ground signals with no voltage loss and pFETs conduct power signals with no voltage loss. CMOS designs therefore use nFETs to control low signals and pFETS to control high signals.

The fourth terminal of the nFET, labeled "substrate", is a connection to the base material on which the FET is fabricated (for an nFET, this is p-material — silicon with an electron deficiency yielding a net positive charge). In digital CMOS circuit design, substrate terminals are always connected to ground (for nFETs) or power (for pFETS),



Some layouts and their fill patterns have been rotated ninety degrees.

Figure 4.8: CMOS layer key.

and will not be included in future transistor diagrams.

Figure 4.7b shows a CMOS inverter, the basic circuit of the CMOS logic family. When the input In is high, the n-transistor pulls the output $\overline{\text{In}}$ low and the p-transistor is off. An analogous case holds when the input is low. This gate will use no power except when the input signal is changing levels, at which time power and ground will be briefly connected as one transistor switches on and the other switches off; during static operation, neither the gate node nor the output node (which is connected to other perfect transistor gates) requires a flow of current.

What has just been described is an ideal circuit, but CMOS transistors are not quite so perfect and do not pass voltages perfectly. The ideal model is, nevertheless, close to the truth and sufficient for most aspects of digital circuit design.

CMOS circuits are formed by placing layers of various materials on top of one another. The major abstract layers for B-SYS, manufactured in a typical CMOS technology, are shown in Figure 4.8. There are five basic layers: polysilicon, two metal layers, and two types of diffusion. Polysilicon is the gate material of transistors: whenever a line of polysilicon crosses either diffusion layer, a transistor is formed. All other layers do not interfere with each other unless connected. For example, polysilicon and both metal layers may pass on top of one another without any electronic effect. The first metal layer (metal1) is the primary connection medium: it can be attached to all other layers using contacts or, in the case of metal2, vias. The second layer of metal, which can only be connected to the first, is well suited for control signal and other long-distance routing. Substrate contacts, usually ignored in transistor diagrams, are crucial in the design of functioning circuits. Beyond these basic layers, more advanced (and more costly) technologies may have additional layers of metal or polysilicon.

A symbolic layout of the CMOS inverter (using only one layer of metal) is shown in Figure 4.7c. Information pertaining to the electrical and switching characteristics of such CMOS gates as well as the use of substrate contacts may be found in the literature.[5,158,151]

B-SYS was designed in a lambda-based scalable CMOS style.[e] In scalable circuit design, layouts are made using an abstract unit of measurement (λ) and various design rules, such as polysilicon having to be both $2\,\lambda$ wide and spaced a similar amount from other polysilicon. The general figure of merit for a fabrication process is its *feature size*, or the smallest size of material which can be reliably fabricated. In abstract units, this corresponds to a $2\,\lambda \times 2\,\lambda$ square. Thus, when B-SYS was implemented in a 2 micron CMOS process, λ was set to 1 micron. The great advantage of scalable CMOS is that all parts of a digital system (excepting pads, which require a certain physical size that does not depend entirely on feature size) may be designed without concern for the eventual technology — the basic functional unit and register bank of B-SYS were designed well before the final implementation technology was known. The inverter in Figure 4.7c could be implemented in a $0.6\,\mu$m process as easily as a $4\,\mu$m process.

The University of California at Berkeley's Magic VLSI design system (both version 4 and beta version 6) was used to lay out the B-SYS chip.[118] A scalable CMOS technology specification for Magic, obtained from the MOS Implementation Service (MOSIS, the fabricator of B-SYS), provided rules for the Magic design rule checker. The design rule checker, running constantly during design sessions, highlights rule violations for correction, a great aid to the circuit designer.

The circuit layouts in this chapter are plots of symbolic Magic designs.

4.2.2 Overview of the B-SYS Chip

The B-SYS chip (shown Figure 4.9 without connecting wires) measures 6.9 mm × 6.8 mm. Each of the 84 square pad cells ringing the chip measures $200\,\lambda \times 200\,\lambda$, or $100\,\mu$m square. These are so "large" both because powerful (hence, large) inverters are needed to drive signals off of the chip and because wires must be bonded from the chip's pads to the pins in the packaging. The pads, obtained from MOSIS, were designed for use with Magic by Charles Seitz at the California Institute of Technology.

The B-SYS chip contains 47 functional units, each coupled with its west register bank (identical to the structure shown in the expanded processor of Figure 4.9), and one additional register bank. Each row of the chip has its own control circuitry, including a finite state automata (FSA) for control signal generation and two decoders, one for the 16-word register banks and one for the eight flags. Also visible are row buffers for the 16 bits of function information associated with each instruction. The entire chip has 85 000 transistors, a modest size in these days of million-transistor chips.

Because sending information on- and off-chip is slow, it is not practical for end functional units to obtain ALU operands from either adjacent chips or the board. For this reason, the register bank shared between the rightmost processing element of one chip (block 47 in Figure 4.9) and the leftmost of the next (block 1) is duplicated. This performs the function of a write-through cache so that chip I/O does not take place during the loading of instruction operands. Whenever a result is written to the right, the value is sent both to the shadow register bank (above the B-SYS logo) and off-chip.

[e] Lambda-based design rules were introduced by Mead and Conway.[120]

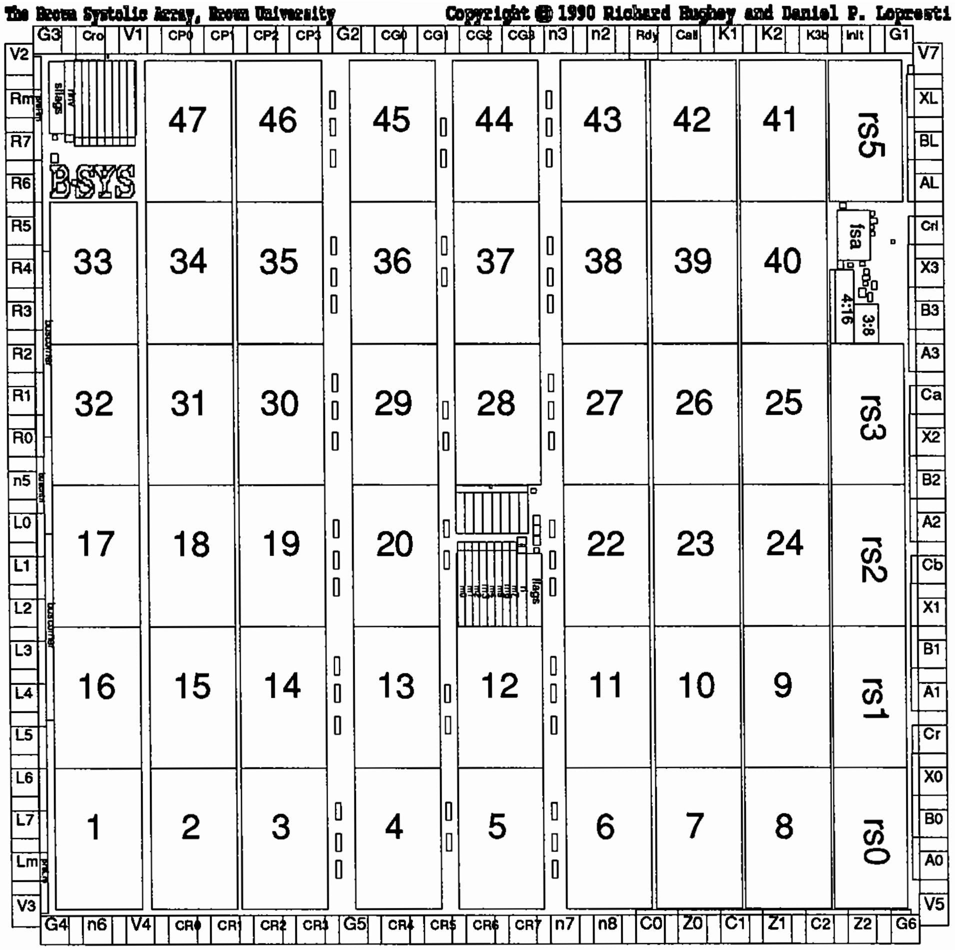


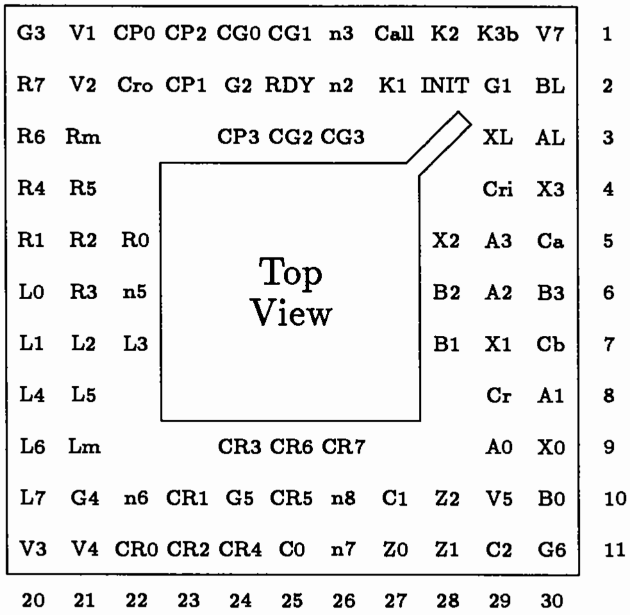
Figure 4.9: Floor-plan of the Brown Systolic Array.

At the same time, an input is received (either from an adjacent chip or the board) for the leftmost register bank (part of block 1 in Figure 4.9).

The mapping of the pads to the 84-pin pin grid array (PGA) is shown in Figure 4.10. The pins have the standard spacing of 0.1 inches, and the square package measures 1.2 inches on each side (about four times the length or width of the actual chip).

The meanings of the pin names are presented in Table 4.4. The pins may be grouped into five categories: clocking (4), instruction (38), data (9 for each side, including a control line for masking), diagnostic output (6), and source voltages (12). The remaining 6 pins are unused. Instruction signals correspond exactly to those described in Section 4.1.1.

Of the four signals in the clocking group, three are actual clocks: K1, K2, and K3. These roughly correspond to decoding the memory address and precharging memory (or evaluating the carry chain), accessing the register bank, and clearing the register selection. There are three primary states the chip can be in: P_a, P_b, and P_{r+i}. During



G3	V1	CP0	CP2	CG0	CG1	n3	Call	K2	K3b	V7	1
R7	V2	Cro	CP1	G2	RDY	n2	K1	INIT	G1	BL	2
R6	Rm			CP3	CG2	CG3			XL	AL	3
R4	R5								Cri	X3	4
R1	R2	R0						X2	A3	Ca	5
L0	R3	n5						B2	A2	B3	6
L1	L2	L3						B1	X1	Cb	7
L4	L5								Cr	A1	8
L6	Lm			CR3	CR6	CR7			A0	X0	9
L7	G4	n6	CR1	G5	CR5	n8	C1	Z2	V5	B0	10
V3	V4	CR0	CR2	CR4	C0	n7	Z0	Z1	C2	G6	11
20	21	22	23	24	25	26	27	28	29	30	

Figure 4.10: B-SYS pin grid array.

Pin	Description	I/O	Pin	Description	I/O
K1	Clock	I	CP0–CP3	Propagate Control	I
K2	Clock	I	CG0–CG3	Generate Control	I
K3b	Clock	I	CR0–CR7	Result Control	I
Init	Ignore Clocks	I	A0–A3, AL	*A* Operand Address	I
C0–C2	*C* Flag Input Address	I	B0–B3, BL	*B* Operand Address	I
Z0–Z2	*Z* Flag Output Address	I	X0–X3, XL	*R* Result Address	I
R0–R7	Right Result Bus	I/O	L0–L7	Left Result Bus	I/O
Rm	Right Write Control	I	Lm	Left Write Control	O
Call	Ignore mask flag	I			
Ca	Phase A	O	RDY	Latch new instruction	O
Cb	Phase B	O	V1–V5,V7	Power	—
Cr	Phase R	O	G1–G6	Ground	—
Cri	Phase R or I	O	n2,n3,n5–n8	Null pad	—
Cro	Phase R or I (slow)	O			

Table 4.4: B-SYS pin names.

the first two states, the A and B inputs are latched in each ALU. During the last phase (P_{r+i}), the results R and Z are calculated and either stored or discarded, depending on the masking conditions. As is perhaps indicated by its name, this phase is longer than the others to allow time for receiving and storing values from adjacent chips. The chip then moves to the neutral P_i state, where it remains until signaled (with the init line) to move on, allowing the host to set up the next instruction.

The data outputs from the execution of an instruction are first available during P_{r+i}

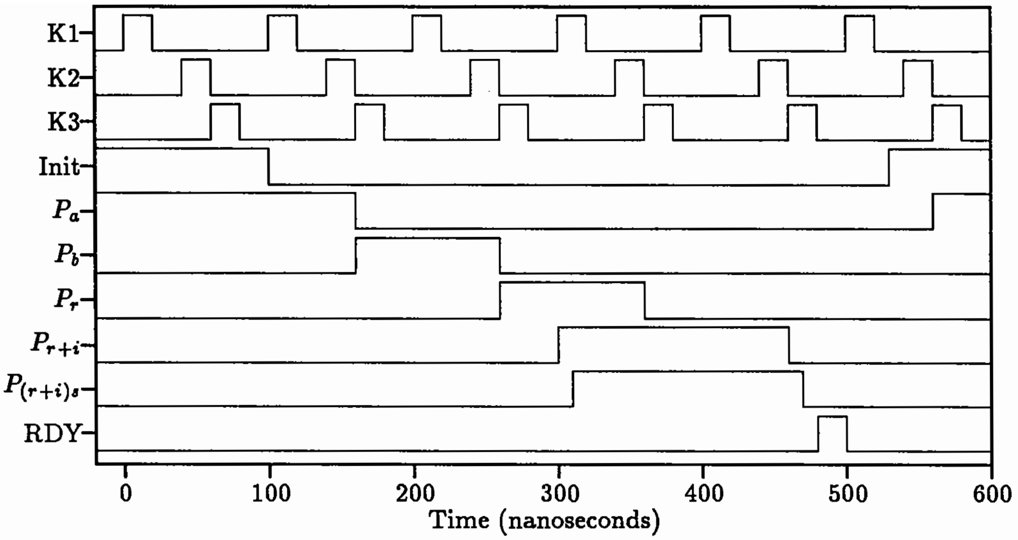


Figure 4.11: B-SYS clocks and diagnostic signals.

and remain until they decay or the next P_a is entered. To aid board design, the chip also provides output of P_a, P_b, P_r, P_{r+i}, a "slow" P_{r+i} for use with edge-triggered latches, and a Rdy signal. The Rdy signal is briefly asserted at the end of P_{r+i} and can be used to control the latching of a new instruction. Figure 4.11 illustrates the clocking just described. During the first cycle of three clocks (K1, K2, and K3) the init signal was asserted, preventing B-SYS from entering P_b until the end of the second cycle. As can be seen, the K2 and K3 signals are allowed to overlap.

The layout of a functional unit and register bank pair is shown in Figure 4.12. In addition to the major features to be discussed shortly (the register bank, function blocks, carry chain, flag circuitry, left and right register bank selectors), several control signal buffers can also be seen. This chapter closes with the completely expanded layout of a functional unit and register bank pair on page 74.

Because of the simplicity of the SSR architecture, the 1730 transistors of a functional unit and register bank pair are densely packed in a $999\,\lambda \times 607\,\lambda$ cell. A larger chip with a smaller technology (B-SYS used $\lambda = 1\,\mu$m) could fit 500 or more functional units and register banks on a single chip, producing an even more powerful single-chip systolic co-processor.

4.2.3 B-SYS Subsystem Design

Many of the B-SYS subsystems are based on circuits found in the literature, in particular the books by Annaratone and by Weste and Eshraghian.[5,158] The units have been designed so that control signals can pass over them from left to right in the second metal layer. This design style is crucial for easily forming large arrays of processing elements:

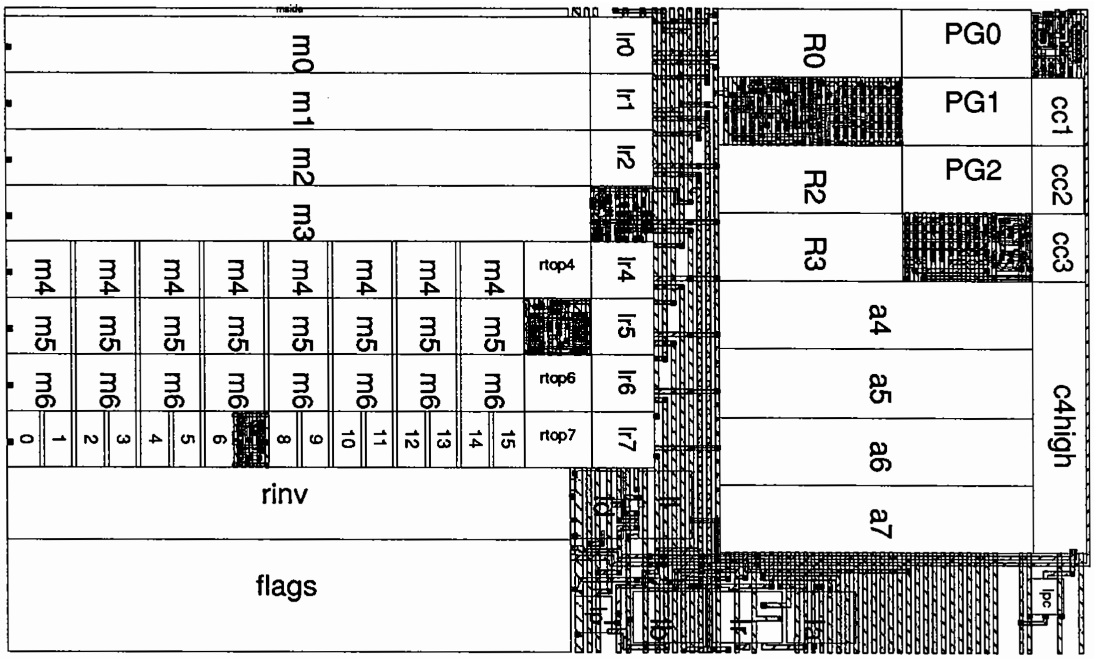


Figure 4.12: Block diagram of functional unit and register bank (partially expanded).

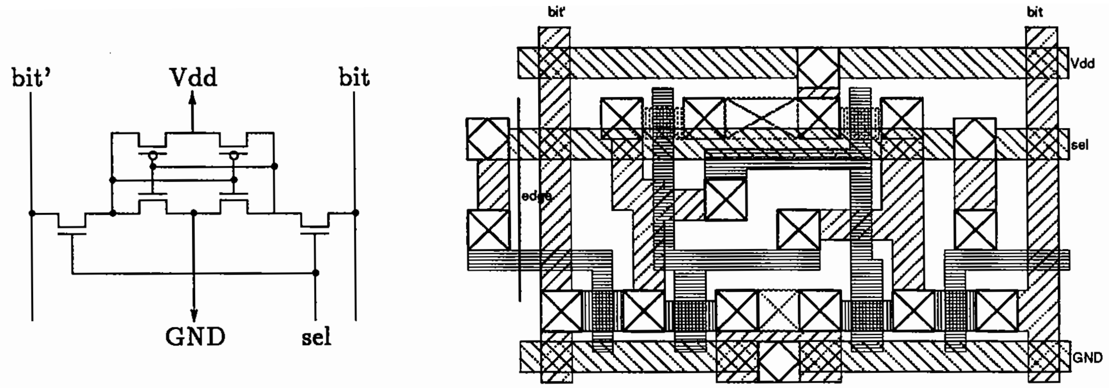
the basic cells can simply be abutted to form all necessary connections. In practice (Figure 4.9), the cells are not quite abutted to allow room for control signal buffers and supply voltage connections.

Subsystems (and the entire chip) were simulated with Carnegie Mellon University's COSMOS (COmpiled Simulator for MOS) switch-level simulator and the Crystal timing analyzer.[14,129] It must be noted that Crystal is a notoriously pessimistic timing analyzer, and thus all Crystal results are loose upper bounds. These two tools guided all aspects of the layout.

Register Banks

Storage elements, both the flags and the registers, use a standard six-transistor CMOS static (without a refresh cycle) random access memory (RAM) cell. The cell, a pair of cross-connected inverters, is shown in Figure 4.13. To read a value from the cell, first the bit and the bit' lines are precharged to Vdd. Then, the desired cell is selected with the sel line. Depending on the value stored, one of the bit lines will be pulled low while the other will remain high. The reason for precharging is size: nFETs do not pass high voltages well, so either precharging or the inclusion of two additional pFETs and the complement of the selection control signal is required. Since the latter choice would significantly increase the size of the RAM cell without as significant a decrease in access time (pFETs are slower than nFETs), precharging is used.

To store a value in the cell, one of the bit lines is raised and the other lowered. When



Ram cell ends at 'edge' marker. A locally unused metal2 line has been eliminated.

Figure 4.13: RAM cell and layout.

the select line is raised, the cell will flip to store the new value.

At the top of each column of RAM cells (Figure 4.12) is control circuitry for the 16-element column: logic for precharging during the precharge clock K1 of P_a and P_b and for writing to the RAM cell during P_{r+i}. Above this is the left or right selection block which connects a register bank to the appropriate functional unit during each phase of instruction execution.

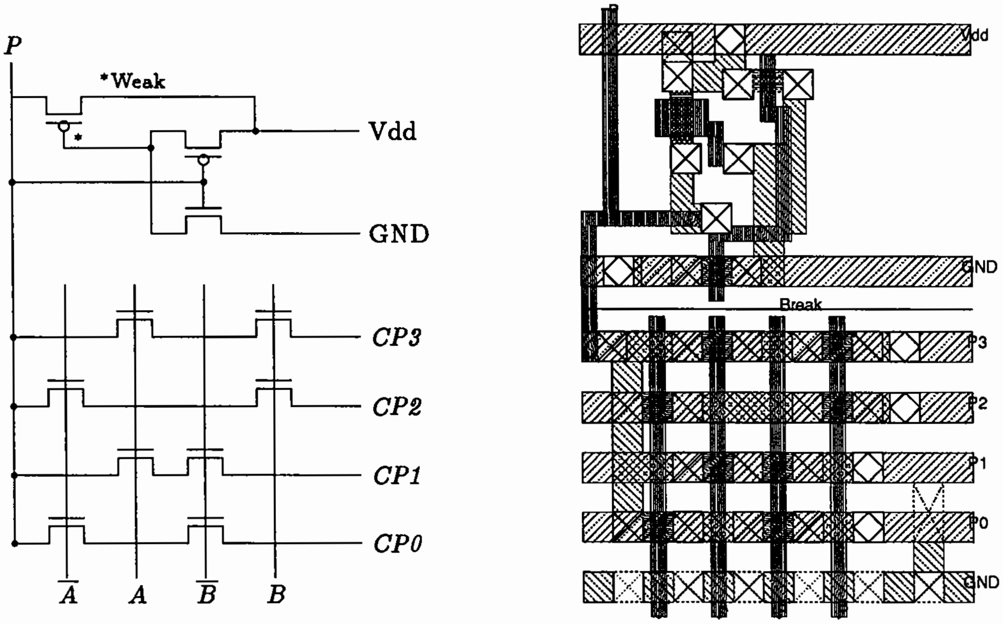
Through extensive experimentation with Crystal, optimal precharge transistor sizes (the larger the transistor, the faster the precharge but the slower the signalling of the precharge) and write buffer transistor sizes were determined. The use of a sense amplifier, an analog CMOS circuit which can shorten the read access time of RAM circuits, was found to yield little improvement for B-SYS' small register banks. Crystal estimates that the register precharge time is 20 ns, the evaluate time is 29 ns, and the write time is 25 ns.

Flags

The eight flags are similar in design to the 16-element RAM columns of the register banks. A selected flag is latched into the C register during P_b, and the Z flag is written during P_r. The flag block of Figure 4.12 also controls access to the RAM cells. The first flag (F_0), the Call control line (corresponding to the '!' field of the instruction word in Figure 4.3), and the Cri signal are used to determine whether or not a memory write should take place, indicated with the Cm control line. Cm activates memory writing in both the register banks and the flag block.

Function Blocks

The implementation of an arbitrary function block is illustrated in Figure 4.14. For any combination of a_i and b_i values, a single control line (CP0, CP1, CP2, or CP3) is selected. The value on this line is then used as the logic block's result P. As has been mentioned,



Eliminated from the layout are the *G* logic block ('Break') and its pull-up network (top right).

Figure 4.14: Propagate logic block.

nFETs do not pass high voltages well. To speed evaluation in this logic block, a weak pull-up pFET is used. Because it is weak, a low signal from a control line will be able to dominate the circuit and turn off the pFET. Although precharging could have been used instead for the P and G logic blocks, it could not have been used for the R block since the c_i input to the R block is liable to change after evaluation has commenced. Precharging will not work unless circuit inputs are constant during evaluation. Special clocking could enable the use of precharging, but this solution is deemed better. According to Crystal, the R logic block requires 27 ns to evaluate after receiving an input (a_i, b_i, or c_i), while the P and G blocks require 15 ns to respond. Changes in control inputs at the chip pads require 23 ns for P, 21 ns for G, and 35 ns for R to stabilize. Since control words are constant for each individual instruction, these times are not important because they are all shorter than the duration of P_a.

ALU Latches

The bottom of each ALU bit-slice includes latches for the a_i and b_i inputs, inverters to provide their complements, precharge transistors to quicken register bank evaluation, and a tristate buffer for the R value. The R logic block and these latches are shown in Figure 4.15. This figure is also an excellent illustration of the design style: control signals are passed over the local logic in a regular grid. In each register bank and functional unit pair, 98 control, data, and supply voltage lines pass over all eight ALU and RAM bit-slices, all but ten in the second metal layer. Given the spacing and width requirements

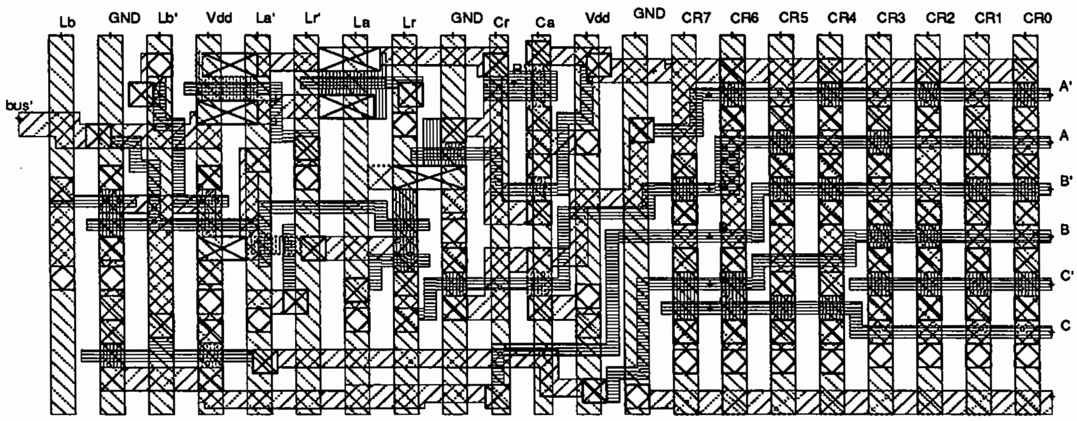


Figure 4.15: *R* logic block and operand latches.

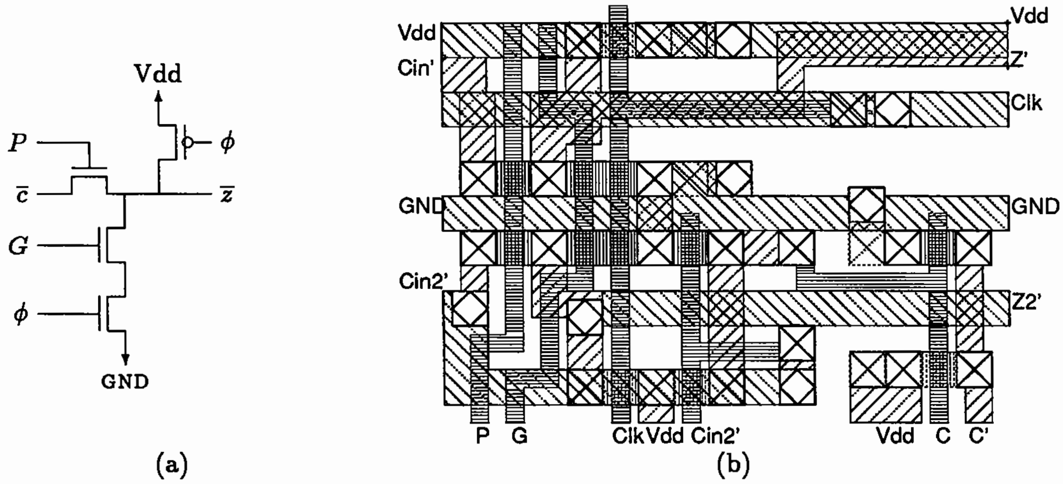


Figure 4.16: One-bit carry block.

of this layer, 124 is the maximum practical number of wires which could cross over the cell, though this would eliminate any use of metal2 for local connections. Both transistor logic and communication and control signals are densely packed throughout the B-SYS chip.

Manchester Carry Chain

Instead of the traditional carry lookahead tree, B-SYS uses a Manchester carry chain, first proposed by Kilburn and others in 1960 when a 24-bit bipolar transistor adder could evaluate in 200 "millimicrosec".[79] The basic cell is illustrated in Figure 4.16a. It computes complemented carry signals, so that during the evaluate phase (ϕ is high), the output $\overline{z}$ will be grounded if there is a generate signal. Otherwise, the input $\overline{c}$ is passed

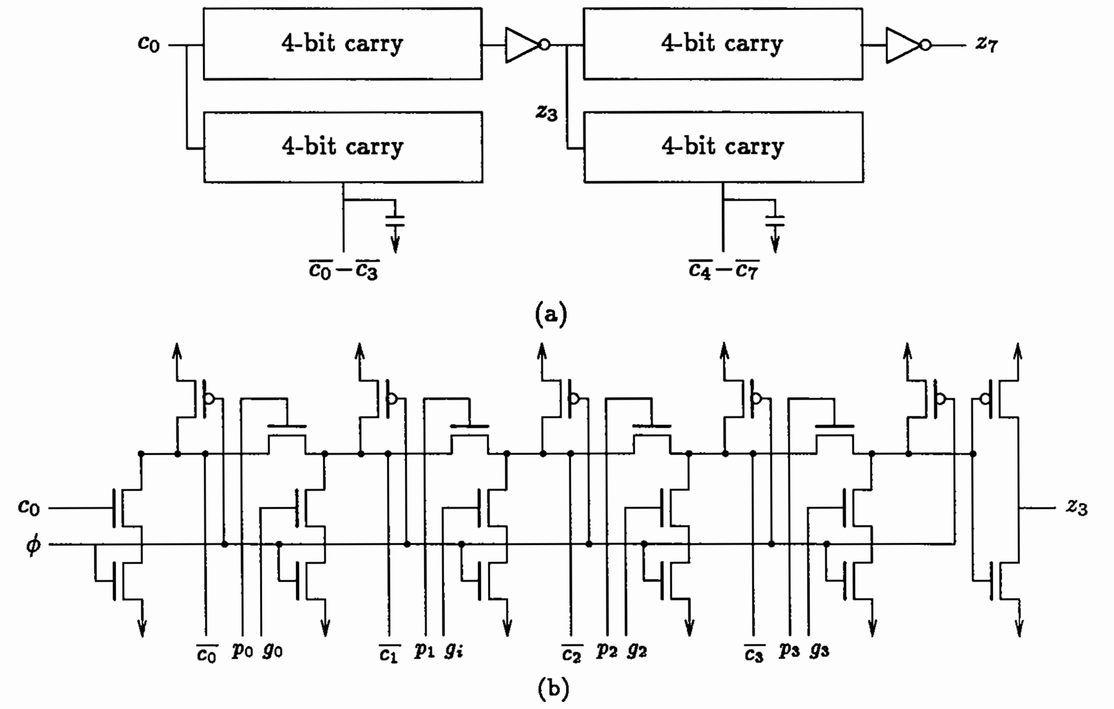

c_0
4-bit carry
4-bit carry
4-bit carry
4-bit carry
z_3
z_7
$\overline{c_0}-\overline{c_3}$
$\overline{c_4}-\overline{c_7}$
(a)
c_0
ϕ
z_3
$\overline{c_0}$ p_0 g_0
$\overline{c_1}$ p_1 g_i
$\overline{c_2}$ p_2 g_2
$\overline{c_3}$ p_3 g_3
(b)

Figure 4.17: The Manchester carry chain.

if there is a propagate signal or the precharged value of $\overline{z} = 1$ remains after evaluation.

As can be seen from the layout (Figure 4.16b), a double carry chain is used. The top carry chain propagates the carry as quickly as possible from c_0 to z_3 to z_7, while the bottom carry chain drives the R logic blocks in the ALU bit-slice (Figure 4.17a). The top z_3 is used to drive both the top and the bottom c_4, an arrangement suggested by Weste and Eshraghian.

A transistor diagram of a 4-bit segment of the carry chain is given in Figure 4.17b. The input is sent through a dynamic inverter and the output is buffered. This electronic separation of each set of four carry cells is necessary because of the large capacitance present in series chains of transistors.

As an early experiment, a ripple carry ALU was tested for B-SYS. In this design, the carry is computed not by a special carry chain but by another 3:1 arbitrary function logic block. Thus, the i-th logic block must be evaluated before the $i+1$-st logic block. Crystal experiments with this ripple carry adder show a carry propagation time of 400 ns. The Manchester carry chain requires less than 5 ns to evaluate.

Control Circuitry

The row control logic has two major parts: the control signal generator at the top (the right in the rotated Figure 4.18) and the decoders at the bottom (the left in the rotated figure). The clocks and input addresses pass from the top to the bottom of the cell so

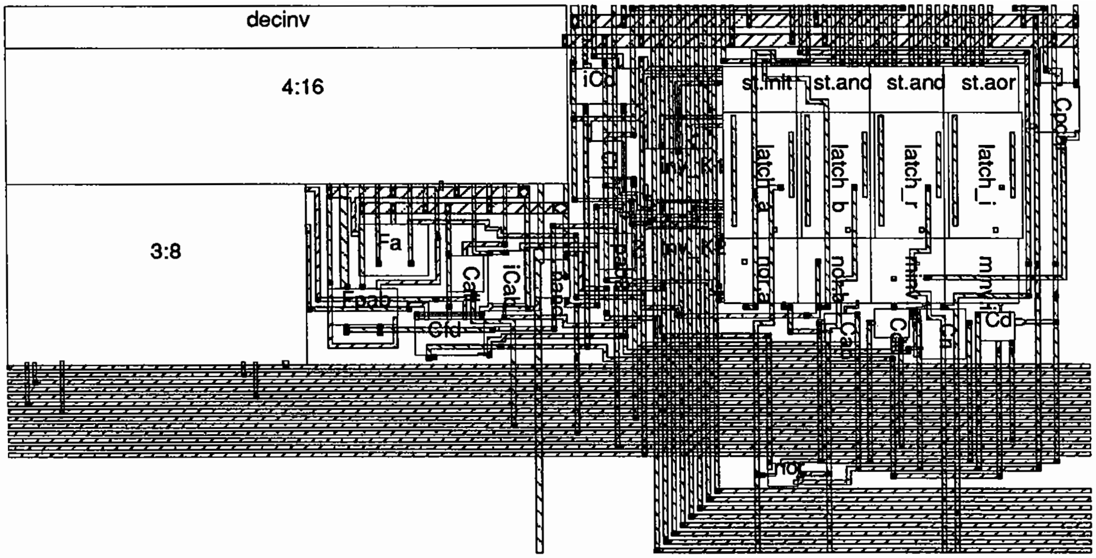


Figure 4.18: Block diagram of the control circuitry.

that rows can be vertically abutted. Also visible are several control signal buffers and the 8-bit data bus from the adjacent row of processing elements turning to meet the next row.

The FSA is of simple design: four static latches, one for each state (from the bottom, P_a, P_b, P_r, and P_i), four next state decoders to the left, and four output decoders to the right. The next state decoders latch an input value during K1 which is then stored in the static state latch during K3 (this is the value seen in Figure 4.11). The value is made available for control signal generation by the output decoders during the next K1. The Ca and Cb control signals (which control the A and B operand latches of the ALUs) are asserted only during the K1 and K2 clocks. The state output decoders for P_r and P_i are more complicated, asserting the Cri line during K1 of P_r and maintaining it until the end (K3) of the first P_i. The intermediate values, latched during K3 in the state latches and connected to the edge of the control cell with the second metal layer, are used to speed address translation.

Decoders

The decoders are based on those of Annaratone.[5] Address decoding commences during K3 when the register selection of the previous phase is cleared. At this point, the address to decode (A, B, or R for register banks) is selected according to the intermediate values of the FSA mentioned in the previous section. The decoders shown in Figure 4.18 continue the address decryption during the precharge (K1) part of the current phase. The inverters (labeled decinv) allow the decoded value to pass to individual register banks during K1, where the signal is buffered again. The value is not passed to the individual RAM cells until K2 since otherwise data would be corrupted during the RAM

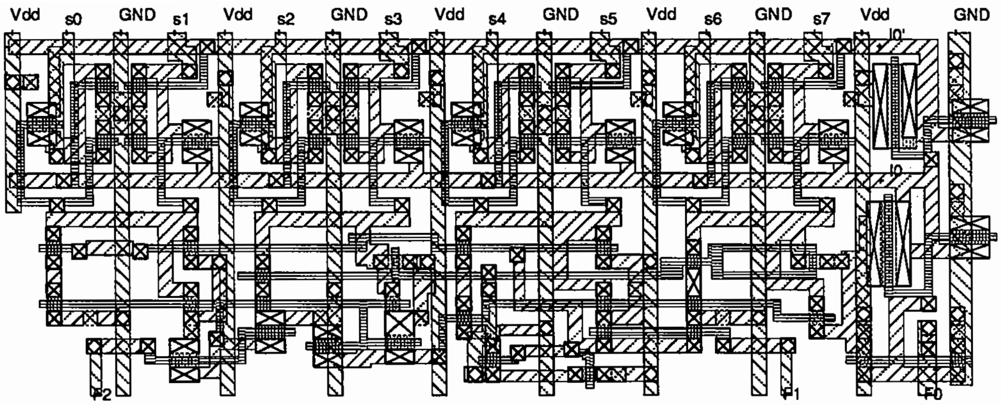


Figure 4.19: The flag address decoder.

precharge. The decoder stops driving the address line at the end of K2, long enough for the register banks to dynamically latch the value. The individual register banks will clear the selection during K3.

The purpose of this scheme is, of course, speed. Decoding register addresses and passing the values to register banks takes a relatively long time (89 ns). By controlling the flow of information (when to make a selection) as close to the RAM cells as possible, a quicker response time is available (34 ns).

The layout of the flag decoder is shown in Figure 4.19. The address inputs are F0, F1, and F2. The selection signals s0–s7, seen at the top of the figure, are buffered, enabled, and cleared in the row control block before being passed to the processing elements. In contrast to the register bank selection signals, this is feasible because flag selection lines only drive one flag in each of the eight flag blocks of the row.

Signal Buffering

One of the most important aspects of chip design, as opposed to subsystem design, is signal buffering. Without the consistent use of sufficiently strong buffers with equal rise and fall times, chip performance will be severely lacking. All control signals are generated locally to each row and then passed to the eight ALUs of the row. All eight functional units are driven by the same control signal, each buffering to eight gates. Thus, the row controller passes the control signal to eight buffers which then distribute the signal locally to 64 bit-slices. Typically, the local control signal is generated 20 ns to 30 ns after the clock (K1, K2, or K3) assertion. Earlier experiments in ripple buffering with in-line inverters produced 70 ns to 90 ns delays, provoking this design switch to faster buffering.

Input and Output

Inputs to the B-SYS chip drive at most six gates, one for each row. These are low-capacitance (minimum size) gates except in the case of the *CR*, *CG*, and *CP* control signals. Since a lengthy setup time is available for all these control lines (P_a and most of

P_b), the increased load does not degrade performance. Outputs from the B-SYS chips use buffer chains of increasing size to drive the large inverters present in the output pads. Driving a signal off-chip through this buffer chain requires approximately 50 ns according to Crystal. Experimental results show this to be much closer to 15 ns than to 50 ns.

The data buses, both left and right, are more complicated. Tristate I/O pads from Charles Seitz' pad set are used. Input is enabled on the appropriate side when an end processing element's local Lri line is asserted, while output is enabled a short while before this using a row's Cri signal. The delay for input enabling is to assure correct treatment of the mask bits. The masking bit for the rightmost processing element is read from off-chip, and this control setup ensures that the correct (non-transient) value of the mask bit is used. In a B-SYS implementation without a mask-bit anomaly (see the footnote on page 50), both the left and the right mask inputs and outputs (depending data flow direction) would be treated this way.

4.3 Design Verification

Logic designs are, of course, worthless without verification. This section briefly considers the logic simulation, timing analysis, and power estimation used to design the Brown Systolic Array chip.

4.3.1 Logic

Logic simulation proved to be an early stumbling block in the design of the Brown Systolic Array because many switch-level simulators are unable to simulate the basic 6-transistor static RAM cell described in Section 4.2.3. Initial simulation used the esim simulator, part of the Magic (version 4) design suite.[118] Discovery of the RAM problem led to the use of Slic, written by Alexander Sherstinsky at the MIT Microelectronics Center. Although this simulator correctly handled the RAM cells, it did not handle indeterminate signals well (for example, an OR gate with one asserted input and one indeterminate input would have an indeterminate output). The third and, thankfully, final simulator used was COSMOS from Randal Bryant and others at Carnegie Mellon University.[14]

COSMOS is a compiled logic simulator: it generates C and assembly language code allowing the circuit to be run as a program. Although this compilation can make the analysis of simple circuits long and tedious, the performance for large circuits is well worth the trouble. Each subsystem was individually simulated, as were entire rows of eight functional unit and register bank pairs. Chip simulation was performed with only four processing elements in the chip at a time: the remaining ones were replaced with stub modules that passed control and data signals from one side to the other.

4.3.2 Timing

As has been mentioned, timing analysis was performed with the Crystal timing analyzer, a pessimistic design tool.[129] Because of the use of properly designed buffers, the B-SYS

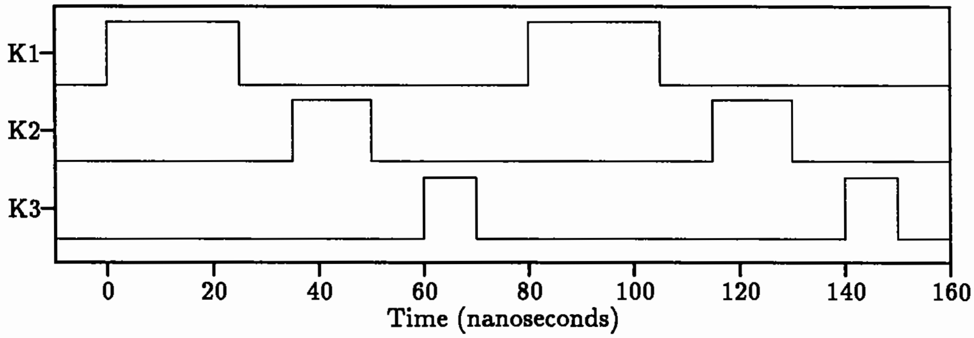


Figure 4.20: Maximum clock speeds from Crystal analysis.

chips can be run *faster* than the worst-case critical path found with Crystal. For example, the first and the eighth processing elements do not need to be executing in lock-step, while the first and the second do need to be reasonably well synchronized. Also, note that all outputs occur far from the row control units, thus inter-chip communication will be reasonably well synchronized.

Applying this principle of allowing small local clock skew (and larger global clock skew) to Crystal timing analysis yields the minimum instruction cycle time of 80 ns shown in Figure 4.20. The dominant requirements are as follows. First, 20 ns are required to precharge memory. Second, 10 ns are required to latch a decoded RAM selection while 30 ns are needed to read a RAM selection (writing is faster). Although only 5 ns are needed to clear a memory selection, 50 ns are needed to propagate the left or right aspect of register selection to memory banks, a task which must be completed *before* entering K2. Finally, an extra 5 ns are allocated to each clock phase to compensate for buffers with slightly unequal rise and fall times.

The actual chips can be expected to perform at this speed, and perhaps at even faster speeds because of the pessimistic nature of the Crystal timing analyzer.

4.3.3 Power

Power estimation is crucial for the sizing of source voltage conductors. From such calculations, the average current densities of wires may be derived. Much the same as the use heavy extension cords with air conditioners, VLSI wires must be appropriately sized according their expected current flow.

In VLSI, pad power requirements are often dominant because of the large drivers needed to send information off-chip. The pads used for B-SYS feature wide supply voltage conductors (50 μm) which ring the chip and are tied to every power and ground pin. Because B-SYS has a relatively small number of output pads, this design is sufficient.

In addition to pad power use, it is important to ensure sufficient power distribution to the main body of the chip, both during static and dynamic operation. The B-SYS chip has negligible static power requirements: in a steady state, there are no direct connections between power and ground and only sufficient current for maintaining gate voltages is required. This is a marked contrast to nMOS and some CMOS design styles which use the MOS equivalent of pull-up resistors.

Dynamic power consumption is the major current sink in CMOS designs. As gates are switched on and off at high speeds, current must be provided to charge and discharge gate and node capacitances. The general equation for dynamic power use is

$$P_d = C_L V_{dd}^2 f, \tag{4.1}$$

which is to say that the dynamic power dissipation P_d is equal to the product of the load capacitance, the square of the supply voltage, and the operating frequency. The average current flow is then

$$I_{\text{ave}} = \frac{P_d}{V_{dd}}. \tag{4.2}$$

For the thumbnail B-SYS power and current calculations, the values $V_{dd} = 5\,\text{V}$ and $f = 1\,\text{MHz}$ were used. Some subsystems switch at higher or lower frequencies, and a corresponding value for f was used. Node capacitances were obtained from Crystal and conservative estimates.

The worst case power consumption of a B-SYS functional unit and register bank pair is $P_d = 7.2\,\text{mW}$, or 342 mW for all 47 blocks in the B-SYS chip. These correspond to average current requirements of 1.44 mA per block and 69 mA for the 47 blocks. Adding the control circuitry and other buffers to this total yields a conservative maximum power estimate of 375 mW or average current of 75 mA for the entire chip, excluding pads. The rule of thumb for conductor sizing is that at most 1.0 mA per micron of metal width should be used. (Conductor sizing is not based on λ units.) For B-SYS' 75 mA, a 75 μm conductor is needed to provide sufficient current to the body of the array. Referring to Figure 4.9, a 44 μm wire is stretched between pads V1 and V4 as well as G2 and G5, a width sufficient for 88 mA. Because the wires are so thick, it was important to position these main supply pads carefully to avoid any bends in the wire (a bend in a 44 μm wire requires $88^2\,\mu\text{m}^2$).

Each B-SYS row has a grid of 17 power lines and 16 ground lines. In the case of ground, the 16 lines have a total width of over 50 μm, well above the requirements for carrying 12 mA of current to each row. Between processing elements, the power and ground lines are tied together to ensure that no supply line carries a disproportionate amount of current.

4.4 Conclusions

This chapter has considered all aspects of the B-SYS design, from the architecture to CMOS layouts to design verification. As has been seen, the design of B-SYS is the result of both interactions between the design levels and experimental testing within levels as the design progressed.

A strict layout style was used to simplify the placement of subsystems on the chip. The Brown Systolic Array's Magic design hierarchy has 100 cells, only 46 of which contain transistors. Although each B-SYS chip has 85 000 transistors, less than eight tenths of one percent of the transistors (658) had to be layed out by hand. Such relative simplicity in the design of powerful co-processors is a prime advantage of the Systolic Shared Register architecture and systolic arrays in general.

The extensive testing and analysis of all parts of the architecture and design, using the B-SIM simulator, a logic simulator, and a timing analyzer, were crucial to creating a chip that was to work on first fabrication.

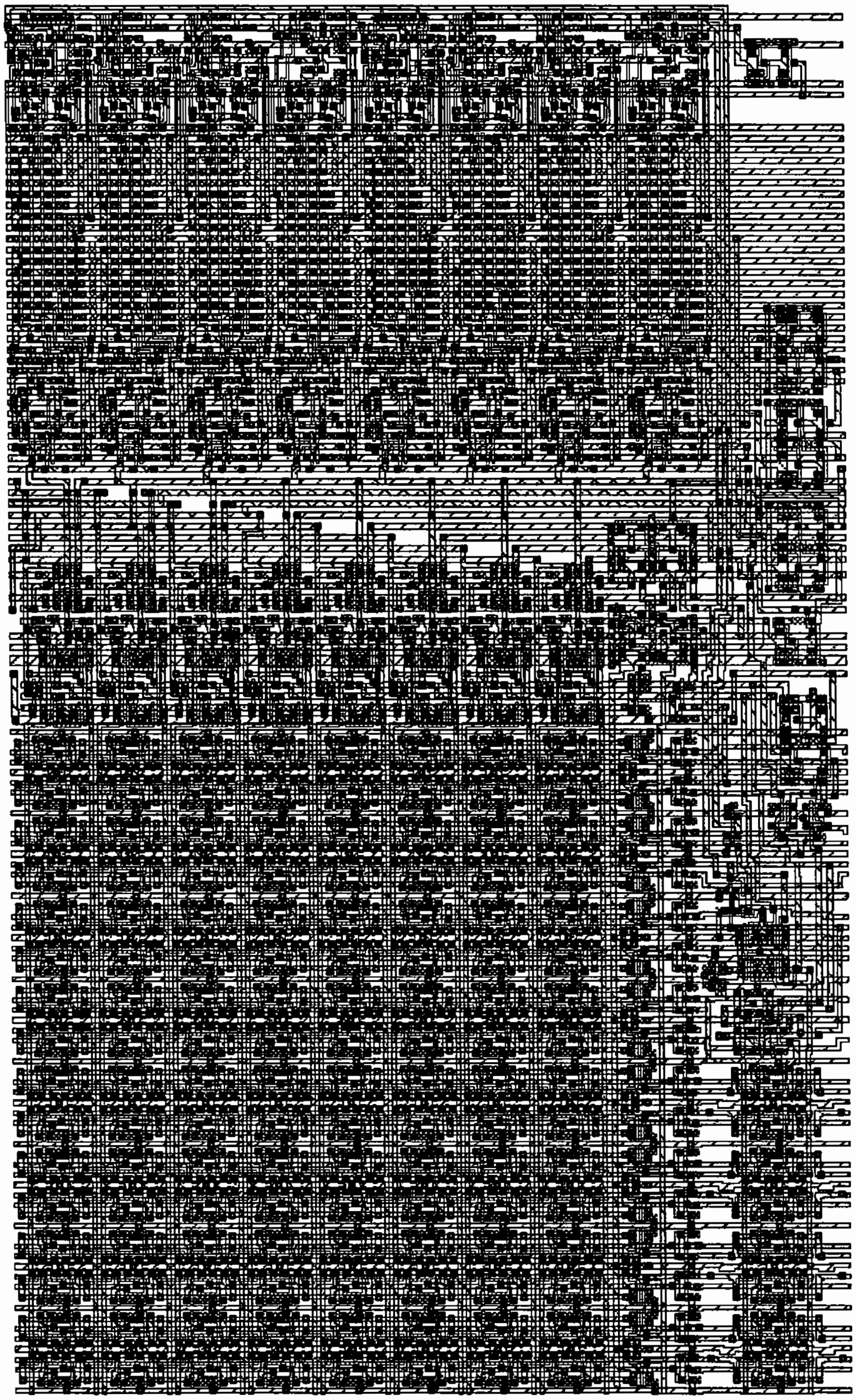


Figure 4.21: Fully expanded B-SYS functional unit and register bank.

Chapter 5

Fault Testing and Fault Detection

FAULTS are an ever-present fact of hardware design, both in the manufacture and assembly of hardware systems as well as in their daily use. This chapter considers first the testing of the Brown Systolic Array chips and the assembly of the B-SYS prototype system. Then, moving to a more abstract level, the problem of transient fault detection during array operation is examined. The principle of software fault detection for programmable systolic arrays, a flexible alternative to hardware methods, is introduced.

5.1 Overview

There are two types of fault detection: off-line and on-line. Off-line testing is used to find permanent faults created by fabrication errors, an unavoidable aspect of VLSI manufacture. This initial screening is performed by shifting the appropriate test vectors into and out of the chip and then checking the results. In the case of the Brown Systolic Array, this was done with the aid of a logic analyzer and pattern generator, as described in Section 5.2.

On-line fault detection takes place as the array is being used, and can find transient (one-time) and intermittent (recurring) faults as well as new permanent faults. Having such a short duration, transient faults are the most restrictive case and thus the most important to consider. Faults can occur in functional units, register banks, and control circuitry; a fault in reading an operand, evaluating an instruction, or writing to memory will corrupt the result.

Before continuing, it is necessary to both define the term "fault" and consider the generic causes of faults. The execution of an instruction is *faulty* if, after execution, the value stored in some register of some register bank and the expected value of that register are not identical. A fault is *detectable* if, in a fault-free computation after the occurrence of the fault, there exist two registers in the neighborhood of a specific functional unit that are unequal as a direct result of the fault. Of course, the assumption of the equality of the two registers is more restrictive than necessary: any form of redundant computation will suffice, such as working with logical compliments, multi-base arithmetic,[31] and shifted operands.[130] In general, the detection method is assumed to be implemented efficiently in the functional units. Because additional faults can corrupt the redundant computation, it is best to detect the fault soon after its occurrence.

Every hardware implementation has its own probable causes of faults and linkages between them. However, there are three basic categories of faults which bear consideration.

1. A fault may be randomly and independently introduced by some external cause (e.g., gamma rays). These are called independent faults.

2. A nearby fault may trigger another fault. For example, an entire register bank or control block could fail, resulting in a large number of faults. These are referred to as space-linked faults.

3. A fault may persist for several clock cycles. These time-linked faults take into account that a fault is not necessarily a single event lasting for the duration of a single instruction. Permanent faults are time-linked faults which last forever, while intermittent faults have higher than normal chances of returning.

Although it is not the purpose of this discussion to provide in-depth probabilistic analysis, it will prove useful to keep these different fault models in mind when considering fault testing and detection.

5.2 Fault Testing

Twenty-four Brown Systolic Array chips were received from MOSIS in the late May of 1990. The chips were subjected to a battery of fault testing routines with the aid of a logic analyzer and pattern generator. Ten chips passed the test procedures and were combined to form the B-SYS prototype system. Despite the prototype's limitations (namely, a slow bus), the 470-element array can perform over one hundred million 8-bit operations per second (100 MOPS).

In addition to describing the test procedure and the B-SYS prototype board, this section examines the performance of a sequence comparison program used as the final stage of the test procedure and considers the design of an ideal board for the Brown Systolic Array.

5.2.1 Chip Screening

In spite of the simplicity of the Systolic Shared Register architecture and the dependability of Magic design, no complicated CMOS circuit can escape fabrication defects. Fortunately, testing an SSR machine is a straightforward task: the regularity of the architecture and its small number of basic units lead to simple test procedures, a great contrast to general-purpose microprocessors. The separation of register banks and functional units implies that, to a large extent, the testing of these two units can be performed separately. Since information is passed into and out of an SSR machine using the register banks, they are tested first. Streaming numbers through the register banks will, in addition to testing the registers themselves, perform minor testing on the functional units and control circuitry. After checking the performance of the register banks, the test procedure moves to a more exhaustive verification of functional units. Each logical component of the functional units (the logic blocks, carry chain, and flags) is individually tested to pinpoint any defects on the chip. The information generated by these tests may then be used to enhance the yield of future designs.

The Brown Systolic Array chips were tested for defects using a Hewlett Packard 16500A logic analysis system, including a 16510A state and timing analyzer and a 16520A pattern generator. The chips were controlled by the pattern generator and their responses verified by the state analyzer. The initial setup buffered all pattern generator outputs to the chip in a manner similar to that of the final prototype board. This setup design motivated by the desire to combine the functionality of the test frame with that of the final prototype board as much as possible.

After confirming that a chip's clock and control circuitry worked correctly using the diagnostic outputs listed in Table 4.4 on page 61, the following tests were performed with the pattern generator and state analyzer:

AA55 The AA55 test shifts the numbers $\mathtt{AA}_{16}$ and $\mathtt{55}_{16}$ through the array (these numbers will detect both bridge faults between adjacent bits and stuck bits). Streaming these numbers through the array in one direction will quickly discover most functional unit, register bank, and control circuit errors. However, no information about the location of the fault (or faults) is available.

InOut The InOut test shifts numbers into the array from one side and then shifts them out again from that same side. Using this test, the location of register bank and functional unit errors can be identified. For example, if there is a fault in $\mathcal{F}_5$ (the fifth functional unit from the west), this will be reflected in five correct results followed by 42 incorrect results when the InOut test is performed from the west side. Testing from both the west and the east side can isolate control errors to specific rows.

Flags The Flags test first sets and clears the flags in each functional unit using the Zzero and Zone control words of Table 4.3 on page 53. Next, the flags are shifted into a working (as determined by the previous tests) register, say W_0, by performing eight additions of the type $W_0 = W_0 + W_0 + F_i$. Finally, these results are shifted out and compared to the expected values.

AddSub The AddSub test also checks the carry chains and the P and G logic blocks. First, one flag is cleared and one register is set to 1. Then, two streams are shifted through the array, one stream having 1 subtracted from it and the other having 1 added to it by each functional unit. The result of this test should be the alternating numbers 48 and 209 ($256 - 47$).

Chip	Comments	Chip	Comments
1	Works.	13	Row 5 stuck low, perhaps caused by bad precharge phase.
2	$\mathcal{F}_9$ bit 2 stuck low. Masking does not work.	14	Works.
3	Right register bank of $\mathcal{F}_5$ does not work.	15	Works.
4	Works.	16	Works.
5	Row 3 does not address F0 correctly, producing a stuck-at-1 fault.	17	$\mathcal{F}_5$ bit 6 stuck low.
6	Control circuitry in row 0 does not work.	18	Output does not work, always high.
7	Works.	19	Row 5 stuck high, perhaps caused by bad decode phase.
8	Works.	20	Undependable masking of $\mathcal{F}_5$ bit 0 and $\mathcal{F}_{34}$ bit 2.
9	Works.	21	Row 5 stuck high, perhaps caused by bad decode phase.
10	Control failure.	22	Control errors in row 2.
11	Works.	23	$\mathcal{F}_5$ bit 0 stuck low. $\mathcal{F}_{10}$ bit 0 stuck high. Error in CR signals or blocks.
12	Flags in rows 0 and 1 do not work.	24	Works.

Table 5.1: B-SYS chip statuses.

Mask The Mask test is, as its name implies, a test of the masking capabilities of the functional units. First, half the context flags are set (odd functional units) and the other half cleared (even). Then, two streams of numbers are shifted into the array. The functional units with asserted context flags copy the first stream to the second, and the resulting stream is shifted out of the array. The test is then repeated, reversing the two sets of functional units.

The state analyzer was used to record invalid output for analysis. By far the most frequent cause of testing errors was chip seating, or ensuring that all 84 pins were firmly connected to the PGA socket. The data busses (R7–R0 and L7–L0) were particularly critical, perhaps because their drivers are less powerful than those of the latch chips, and thus less able force signals over bad connections.

The results of these tests are displayed in Table 5.1. As can be seen, ten of the chips worked correctly. Also, the test suite was able to provide exact information about many of the fabrication defects. It is difficult to draw any conclusions from the faults both because of the small sample size and the lack of information about the positioning of the chips on the wafer. However, all the defects cluster around the outside of the chip (row 0, row 5, the control circuitry, and processing elements close to the edge of the chip, as seen in Figure 4.9 on page 60), possibly implying that the defects occurred in the process of

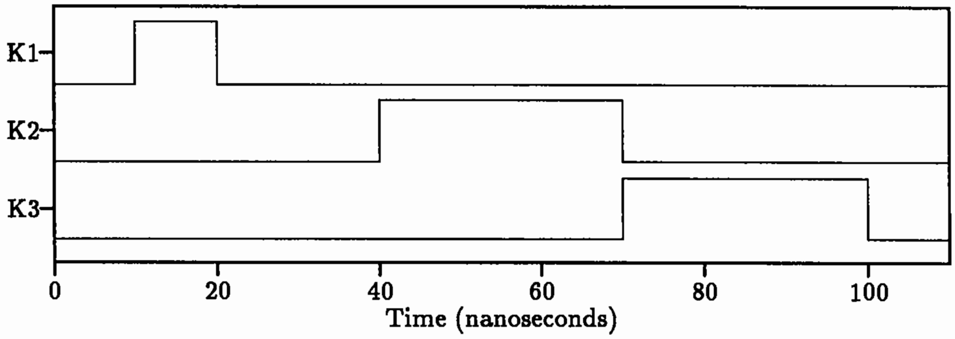


Figure 5.1: Pattern generator clock phases.

cutting the wafer into chips or bonding the chips to the PGA packages. Unfortunately, apart from using much less of the chip's total area, such faults are difficult to avoid by design modification.

The chips were tested with a 100 ns clock cycle time (K1 to K3), as illustrated in Figure 5.1. Because of the pattern generators' timing gradations, 50 ns was the next fastest speed that could be checked, and a speed at which the support logic and timing analyzer combination could not reliably deliver clock signals to the chip. Experimentation with the clock phases at 100 ns seems to indicate, however, that the B-SYS chips can function at considerably higher speeds.

5.2.2 The Prototype Board

A prototype board for the ISA[a] bus was acquired from JDR Microdevices. This board, configured as an I/O device, provides both bus interface circuitry and address decoding.[b] To the board's bus interface, latches for instructions, inputs, and outputs were added, as well as room for 12 B-SYS chips (Figure 5.2, the support logic is itemized in Table 5.2). Headers were used during chip testing to connect the pattern generator outputs to the instruction and input latches. The board's one-thousand connections were made by the author with a hand-held wire-wrap gun.

The B-SYS prototype array uses the ISA bus' I/O addresses 300 through 30C. The first three 16-bit words are for instruction and input (I7–I0) signals to the array while the last is used to read output values (O7–O0) from the co-processor. The meanings of individual bits are shown in Table 5.3.

As the board was assembled (running into many chip seating problems), several test programs were executed to check its operation. In addition to assembly language and

[a] Industry Standard Architecture, IEEE Standard P996, also referred to as the AT bus.

[b] On the subject of address decoding, the board's preprogrammed PAL (programmable array logic) chips did a less than admirable job of interpreting bus traffic. The first set of PALs interfered with bus traffic whenever a read operation to the graphics card took place. A second set of PALs was tried, but this set interfered with disk I/O. The first set was chosen as the lesser evil.

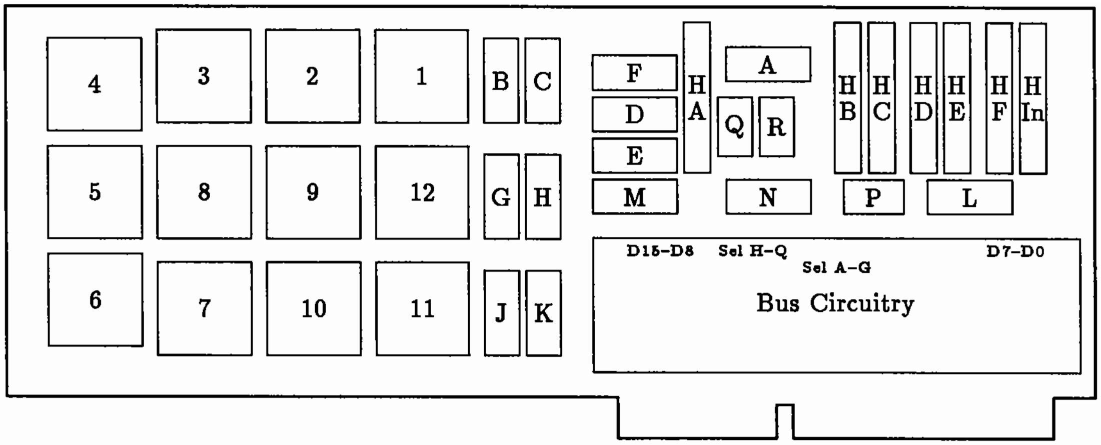


Figure 5.2: B-SYS prototype board.

Label	What	Purpose	Label	What	Purpose
1–12	PGA	84-pin PGA sockets for 12 B-SYS chips.	G	373	Left input latch, enabled by $\overline{\text{XL}}$.
A	373[a]	1-to-2 latch for K1, K2, K3, and Init.	H	373	Left output latch, enabled by XL.
B	373	Latch CG3–CG0 and CP3–CP0.	J	373	Right input latch, enabled by XL.
C	373	Latch CR7–CR0.	K	373	Right output latch, enabled by $\overline{\text{XL}}$.
D	373	Latch XL, X3–X0, BL, B3, B2.	L	373	D15–D8 input latch (high byte from ISA).
E	373	Latch B1–B0, AL, A3–A0.	M	373	D7–D0 input latch (low byte from ISA).
F	373	Latch Call, Z2–Z0, C2–C0.	N	373	D7–D0 output latch (low byte to ISA).
P	00[b]	NAND, control signal inversion.	R	00	NAND, control signal inversion.
Q	00	NAND, enable signal for N.			
HA	Hdr[c]	Clock input header.	HD	Hdr	D header, now tied to L.
HB	Hdr	B header, now tied to L.	HE	Hdr	E header, now tied to M.
HC	Hdr	C header, now tied to M.	HF	Hdr	F header, now tied to L.
			HIn	Hdr	G/J header, now tied to M.

[a] 74HCT373: Tri-state octal D-type latch. [b] 74HC00: Quad nand gate. [c] 10-by-2 header suitable for pattern generator output.

Table 5.2: B-SYS prototype board key.

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
300	CG3	CG2	CG1	CG0	CP3	CP2	CP1	CP0	CR7	CR6	CR5	CR4	CR3	CR2	CR1	CR0
304	XL	X2	X2	X1	X0	BL	B3	B2	B1	B0	AL	A3	A2	A1	A0	—
308	Call	Z2	Z1	Z0	—	C2	C1	C0	I7	I6	I5	I4	I3	I2	I1	I0
30C	—	—	—	—	—	—	—	—	O7	O6	O5	O4	O3	O2	O1	O0

Table 5.3: B-SYS prototype I/O addresses.

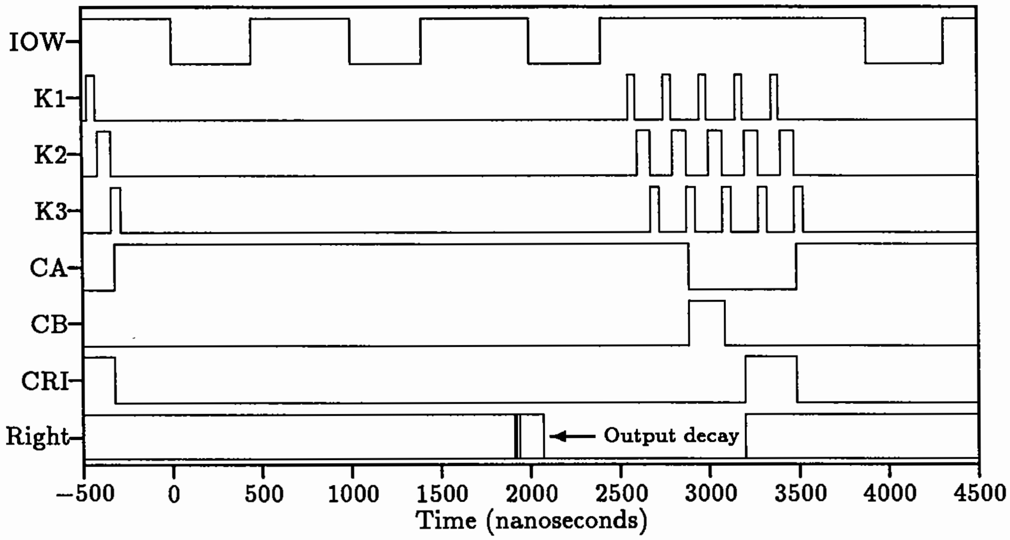


Figure 5.3: Prototype clocking.

C versions of the pattern generator tests, more sophisticated programs, such as sorting and sequence comparison, were tested.

5.2.3 Prototype Performance

Figure 5.3, a transcription of logic analyzer data, illustrates clocking on the prototype system. The clocks are produced by the pattern generator which triggers a set of five clock cycles whenever the third word of an instruction is written to the board (I/O address 308). During the first set of three clocks (K1, K2, and K3) the init signal was asserted, preventing B-SYS from entering P_b until the end of the second cycle. As can be seen, the processing speed of the B-SYS prototype board is overwhelmed by the time required to write three 16-bit words to the prototype board (IOW is the ISA bus' asserted low I/O device write line). Also note that the decay of B-SYS outputs, mentioned on page 62, takes place approximately 2 ms after the end of an instruction.

As the board was assembled, correct execution of a typical application was verified. The sequence comparison problem of Section 2.1.3 was solved both on B-SYS and on

Processors	File Access	C	B-SYS	B-SYS MOPS	Speedup
94	7.6	63.0	4.2	24	15.0
141	9.1	148.	7.9	32	18.8
188	9.8	216.	10.6	43	20.3
235	10.6	336.	13.2	54	25.5
282	33.5	480.	15.8	65	30.4
329	33.6	653.	18.3	76	35.7
376	33.3	856.	20.9	86	40.9
423	33.5	1080.	23.4	97	46.2
470	33.4	1330.	26.3	108	50.6

Times are in milliseconds.
C and B-SYS times ±1%, file access times ±2%.

Table 5.4: B-SYS sequence comparison timings.

the 80386, and then the results were compared (the B-SYS solution will be described in Section 7.1). Timings for this algorithm, as well as the disk access time for loading the sequences, are shown in Table 5.4. In all cases, the routine had to be executed multiple times to compensate for the low granularity of the system clock. For the full array, the C routine was executed 5 times and the B-SYS routine 213 times. It is interesting to note that although the prototype board is slow, it can keep pace with file access to the system's hard disk.

In addition to this algorithm, Chapter 7 analyzes a faster sequence comparison algorithm which performs over 80 times faster than the 80386, as well as a variety of other applications.

5.2.4 The Ideal Board

As is illustrated by the timing diagram (Figure 5.3), the B-SYS prototype board does not provide an efficient interface between the host and the co-processor. The large amount of time spent accessing the board reduces performance to a mere 108 MOPS, or 0.23 MOPS per processing element. This section considers ways to drive the array at full speed, approaching the 4 MOPS per processing element indicated by the optimal clocking scheme of Figure 4.11 on page 62.

Systolic algorithms typically repeat a single cell program many times. Thus, over the course of a computation, only a small set of instructions is needed. This leads to the first refinement: instead of requiring the host machine to access three I/O addresses for each instruction, a collection of instructions could be preloaded to an on-board instruction memory and indexed by the host. To execute a preloaded instruction, the host would send, for example, an 8-bit instruction address and an 8-bit input value to the board, reducing the bus use to once per instruction (twice if operands must be read from the board). Such an arrangement could triple the speed of the co-processor, moving it up to 0.75 MOPS per processing element. In spite of this improvement, the processing speed would still be controlled by the bus access time since the host must access the board to

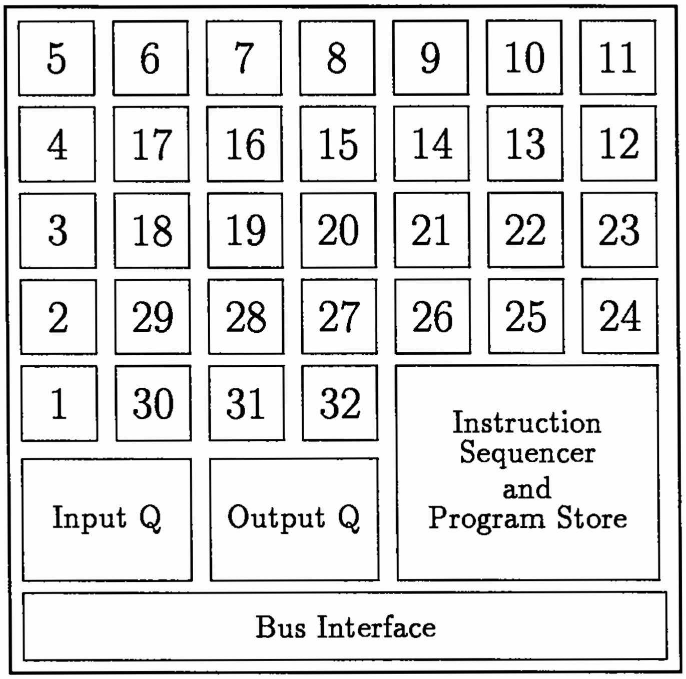


Figure 5.4: B-SYS* ideal board.

execute each instruction regardless of whether or not it must transfer data to or from the board. Performance could be further improved by decoupling the board and the bus.

The final refinement is to include both a program memory and a microsequencer on the co-processor board. Also, to allow asynchronous operation, blocking input and output queues could be used to transfer data between the host and the board. This board, referred to as B-SYS*, is illustrated in Figure 5.4 (note that it is decidedly not an ISA board). Entire routines would be preloaded to the board and could be initiated with a single command. The program, in addition to storing the required instructions, would also have specifications for default inputs (such as zero or the value of an on-board register) to limit the use of the queues. The host could then fill the input queue and remove data from the output queue at its leisure, though it must ensure that deadlocks do not occur. Also, this system would use the B-SYS diagnostic outputs to control instruction, input, and output latching. Such a system could drive the array very quickly, perhaps increasing performance to 2 MOPS or more per processing element. With this setup, a one-board, 32-chip array (1504 processing elements) could provide 3 GOPS of systolic co-processing power.

5.3 Fault Detection[c]

Having described permanent fault testing for the Brown Systolic Array chips, this section turns to transient and intermittent fault detection during the execution of arbitrary systolic programs. Fault detection research has generally focussed on two strategies: hardware fault detection and algorithmic fault detection. Typical hardware fault detection methods duplicate each computation with additional hardware units and compare outputs for discrepancies with additional circuitry. Algorithmic methods focus on the high-level algorithm; matrix operations may be made fault-tolerant if checksum rows and columns are added to the arrays.[66] Such an approach requires no special hardware but is limited in its utility because rigorous mathematical analysis is required for each new application.

This section proposes a third approach: *software fault detection* for programmable systolic arrays. Automatic program transformations for creating fault-tolerant programs are considered, as well as refinements which make use of specific features of the original program. Unlike algorithmic methods, software fault detection does not rely on a mathematical study of the problem being solved. Unlike hardware methods, the user has full control over all available resources (processors, memory, and time). The user can make the computation more robust by allocating additional resources to fault detection or even eliminate the software fault detection if either an alternate type of detection or no detection is preferred.

Thus, software fault detection methods allow a programmable systolic array to be used both when obtaining maximum performance is the primary concern and when fault detection is so important that the user is willing to trade decreased performance for an increased ability to detect faults. Instead of building two separate systems, one optimized for performance and the other for fault detection, software fault detection makes use of existing hardware with no special modifications; the inherent redundancy of the systolic array is used for fault detection. Systolic programs are automatically transformed from their original version to a version able to detect any predetermined number of both permanent and transient faults. Unlike algorithmic schemes, software fault detection is easily automated. The tradeoff is that more robust programs require either more time to execute or more processing elements to execute on.

5.3.1 Hardware Fault Detection

Perhaps the best known approach to fault detection and correction is triple modular redundancy (TMR).[80] In this classic scheme, each computation is performed on three independent hardware units which then vote to determine the result. A generalization of this is d-way modular redundancy (d-MR). For example, 2MR allows only single fault detection without correction. TMR, in addition to more than trebling the hardware cost, will slow down the computation: each comparison or vote adds hardware complexity and computational delays.

[c] Parts of this section were presented by the author at the Second IEEE Symposium on Parallel and Distributed Processing.[70]

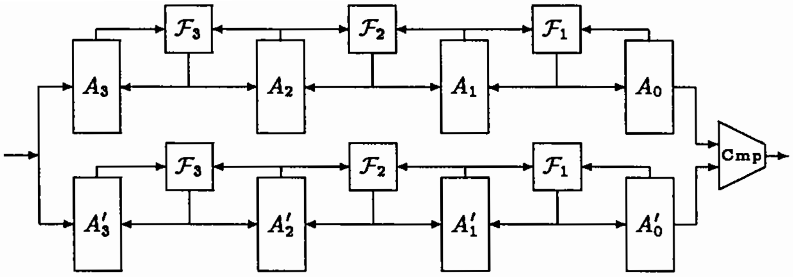


Figure 5.5: Gross replication of computation.

In the roving spares method (RS) proposed by Shombert and Abraham,[141] k extra processing elements independently "roam" through the array checking processors in turn. Since processors do not actually move, each processing element must be able to act as a spare or as an active processor. Roving spares work best with multiple instruction and multiple data stream (MIMD) computation since the spares will be able to act autonomously. Unless there is one spare for each processing element, a transient fault can go undetected.

Token triggered comparison with duplicate data (TTCDD), presented by Choi and others, is more appropriate for machines with either broadcast instructions or no instructions (i.e., systolic machines in which each cell performs some fixed function each and every time unit).[27] In this approach, k extra processing elements must be added to the array as well as extra routing and comparison circuitry for each processing element. As the computation proceeds, a sequence of k tokens is cycled through the array. These tokens determine which computations should be replicated for fault testing. In effect, these are roving spares without autonomy. As with the RS method, unless $k = N$, the size of the original array, transient and intermittent faults can go undetected.

5.3.2 Software Fault Detection

Checksum methods for detecting faults during the execution of parallel matrix operations have recently gained attention.[12,66] By adding checksum entries to matrices, 85% or better fault coverage may be achieved for problems such as matrix multiplication, fast Fourier transform, QR factorization, singular value decomposition, and adaptive filtering.[12] This tolerance is accomplished entirely in software; complicated hardware changes are not required. However, these routines do not apply to general systolic programs but instead result from a careful analysis of a specific computation. This section proposes fault detection methods for arbitrary algorithms. The user is not required to perform a complicated analysis to determine the means of fault detection. Instead, simple schemes which duplicate computation in software are considered.

As an example, consider the code fragment

$$E_0 \leftarrow \min(W_0, W_1). \tag{5.1}$$

Since this statement not only performs a minimization but also moves data eastward,

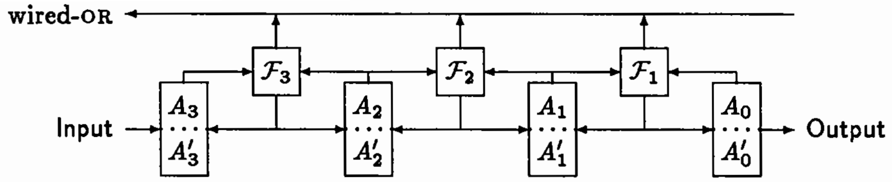


Figure 5.6: Replicated computation streams for fault detection.

making this single statement fault-tolerant will illustrate the process of making entire programs fault-tolerant.

To detect transient faults in arbitrary programs, computation must be duplicated and monitored: without complete coverage, a transient fault can be missed. Since arbitrary programs have no predetermined relationship between the data in various parts of the processor array (as opposed to checksums), full duplication is required. General fault detection thus requires at least twice as much time or at least twice as many processing elements as the original program. With software fault detection, even when twice as many processing elements are used, the actual hardware remains fixed: special routing and comparison circuitry is not required by the method.

Perhaps the most obvious method of software fault detection is to use a gross replication of the computation, shown in Figure 5.5, in which output results are compared as they leave the array. The two computation streams, A and A', should produce identical results — after the execution of each instruction, A_i should equal A'_i for all i. Whenever the results from the A and A' execution are not identical, a fault has occurred. Gross replication of computation requires double the amount of time or, equivalently, double the number of functional units and memory nodes. For all this effort, only one comparison is made during each time step, leaving the potential for many undiscovered faults. If the same hardware is used for both computation streams (time multiplexing), the same time-linked fault could occur in both streams and go undetected. Also, gross duplication of computation will not provide any indication as to where the fault occurred. Locating faults is critical for array reconfiguration and software avoidance; it may be desirable to reconfigure the array to avoid functional units or memory banks with intermittent faults. Without the location, one may as well replace the entire array and hope for the best. Parallel fault detection methods are a vast improvement over this simple scheme.

A better way to detect transient faults on an SSR machine is to replicate the computation stream as shown in Figure 5.6. Two identical (in the fault-free case) computation streams flow through the array at the same time. Since there are two different and neighboring memory locations which, in a fault-free computation, are identical, faults are detectable in the sense of Section 5.1. Comparisons may be made at *all* functional units as frequently as desired, providing a strong method for detecting and locating transient faults. Thus, fault detection can occur close, in both time and space, to the actual fault. For the moment, it will be assumed that functional units have a dependable (faultless) method for reporting faults, such as the wired-OR in Figure 5.6. Taking the statement of Example 5.1, the A' computation stream is assigned the second block of

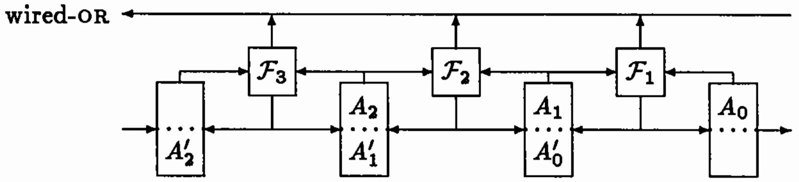


Figure 5.7: Skewed replicated computation streams for fault detection.

four registers (i.e., E_0 is shadowed by E_4). The code is then expanded to process both computation streams and relay the fault information to the host computer:

$$\begin{aligned} E_0 &\leftarrow \min(W_0, W_1) && (5.2.1)\\ E_4 &\leftarrow \min(W_4, W_5) && (5.2.2)\\ \text{wired-OR} &\leftarrow (E_4 \neq E_0). && (5.2.3) \end{aligned}$$

Here, the first instruction (5.2.1) operates on the A (original) stream, while the next instruction (5.2.2) operates on the A' stream. Input values must be duplicated and made available to both the A and A' computation streams.

Most systolic algorithms have both moving and fixed data streams. Since errors in the computation of fixed (local) data will be reflected latter in the moving data stream, it is not necessary to compare entire register banks. Only those registers used by the moving data streams need to be checked (by comparing corresponding values in the computation streams).

One of the problems with the simple duplication scheme just described is the ability for a corrupt register bank to go undetected. If, for example, a fault results in a bit position always being written as a '1' throughout an entire register bank, the A and A' values would be both the same and incorrect. Although computing the A' stream as the logical negation of the A stream would solve this case, bridge faults (in which neighboring nodes are forced to the same value via a connecting bridge) would remain a problem. The solution to this space-linked problem is the *skewed replicated computation stream* (SRCS) method of permanent and transient fault detection (Figure 5.7). The computation is again replicated on two independent computation streams, A and A', however the streams are skewed by one register bank. Faults are still detected by comparing corresponding $A_i - A'_i$ pairs using a broadcast instruction, and when they differ a fault is signalled. Further replication to d computation streams will provide better performance in terms of the worst-case number of detectable faults $(d-1)$.

Using this method, the fault-tolerant version of Example 5.1 is:

$$\begin{aligned} E_0 &\leftarrow \min(W_0, W_1) && (5.3.1)\\ E_4 &\leftarrow \min(W_4, W_5) && (5.3.2)\\ \text{wired-OR} &\leftarrow (W_4 \neq E_0). && (5.3.3) \end{aligned}$$

Of course, this transformation is nearly identical to that of Example 5.2, however the skewing of the computation streams is a critical conceptual point. Again, although

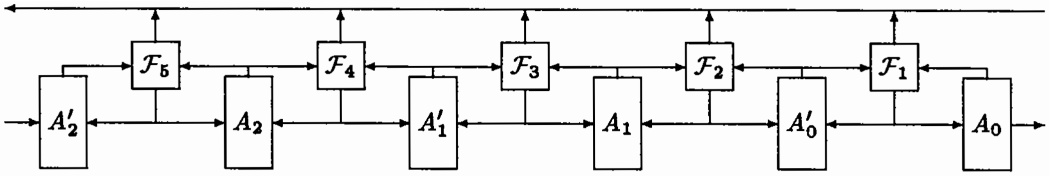


Figure 5.8: Interleaved replicated computation streams.

Example 5.1 expanded to three statements in Example 5.3, SRCS does not generally produce a factor of three slow-down since a systolic cell program with S non-local data streams expands to an SRCS program with $2I + S$ statements, where I is the original number of instructions. For added fault detection (or correction), SRCS can be extended to d computation streams requiring cell programs of size $dI + S(d-1)$.

SRCS has several advantages over the previous two schemes. First, unlike gross duplication of computation, the location of the faulty processor can be determined by host query. Such faulty processors may then be avoided with appropriate software modifications or (if available) array reconfiguration. Second, the skewing of the streams protects the system from catastrophic memory bank failure: since A_i and A'_i are in different memory banks, a corrupt memory bank will not invalidate the method. Also, since A_i and A'_i share a common functional unit, no extra data communication is required for the comparison. If that common functional unit is faulty, an adjacent functional unit will be able to detect the problem when it operates on a different data set (i.e., A'_{i-1} or A_{i+1}).

For programmable systolic arrays with small amounts of memory, the use of skewed replicated computation streams could overwhelm the register banks. The solution to this problem is to use twice as many functional units and register banks, interleaving the A and A' data sets in neighboring registers (Figure 5.8). This method of interleaved replicated computation streams (IRCS) has the flavor of a hardware approach but does not require the use of special circuitry or modified functional units. As one might expect from the use of more processors, an IRCS program will be faster than the corresponding SRCS program. Using IRCS, even functional units process one computation stream (A) while odd units process the other (A'). Reading from and writing to the west register bank takes place as normal (i.e., even processing elements have A values and odd have A' values in their respective west register banks). Accessing the east data set is more complicated since it is no longer adjacent to the appropriate functional unit. For each original access of the east register bank, a SIMD instruction to shift values from east to west (for a read) or west to east (write) must be added to create the fault-tolerant version. Example 5.1 is transformed via IRCS to

$$E_0 \leftarrow \min(W_0, W_1) \tag{5.4.1}$$
$$E_0 \leftarrow W_0 \tag{5.4.2}$$
$$\text{wired-OR} \leftarrow (E_0 \neq W_0) \text{ and } \langle\text{Odd processor}\rangle. \tag{5.4.3}$$

Here, instruction (5.4.2) shifts values throughout the array to the east (recall that W_0

is E_0 for an adjacent processor). For better fault protection, a data check in the even processors similar to statement (5.4.3) may be added after statement (5.4.1). As with SRCS, the number of extra instructions depends on the number of moving data streams in the cell program. Taking this number S into account, the SRCS cell program will have $I + 2S$ instructions. Thus, an SRCS program does not increase the leading coefficient of the execution time, yielding nearly the same performance as the original program.

This method may be extended to d replications of the computation, in which case dN processors are required to execute the cell program of $I + 2S(d-1)$ instructions.

5.3.3 Propagation of Fault Information

There are two problems with using a wired-OR bus for fault reporting. First, indication of the location of the fault is not available. Second, of course, this is a hardware fault notification method. Since it requires the availability of special hardware, it is not useful for some extant arrays, B-SYS among them. The alternative is a data stream for fault notification which, unfortunately, might be compromised by a faulty register bank or functional unit. This section considers methods for generating dependable fault detection streams in software.

First, word-size redundancy in the fault stream can increase its dependability. In the B-SYS case, this would imply that one 8-bit value indicates a correct computation and the 255 other values indicate a faulty one. For example, any non-zero value could represent a fault. Such a scheme will provide some resilience to single-bit errors.

The two preferred methods of the previous section, skewed replicated computation streams and interleaved replicated computation streams, protect against functional unit errors by computing corresponding parts of the two computation streams in different functional units and register banks. If there is only one fault stream, it could be corrupted by a single functional unit or register bank. To provide added security (beyond the word-size redundancy of the error message), two fault detection streams may be used, one traveling in each direction. With two notification streams, fault location can be readily determined without host query: if fault flags exit opposite ends of the array during the same step, the fault occurred in the central functional unit or one of its register banks. Failures in other functional units and register banks can be pinpointed by observing the delay between fault reports in the two fault streams, in a manner reminiscent of the InOut test used to screen the B-SYS chips. Multiple transient faults cannot be located with ease because of the I/O bounds on linear systolic arrays. If only one stream indicates a fault, an error in one of the fault detection streams has occurred.

The use of fault detection streams does not significantly increase the size of the fault-tolerant program. Locally generated fault information on the S data streams of a program can be combined as it is computed. Then, this information can be combined with the fault detection stream as it is shifted though the array, a feature of the SSR architecture. Thus, one instruction per cell program iteration per fault detection stream is added by the use of software fault notification.

Resource	Software Approaches		Hardware Approaches		
	2SRCS	2IRCS	N-RS	N-TTCDD	2MR
Hardware	$N+1$	$2N+1$	$2N(1+\epsilon_1)$	$2N(1+\epsilon_2)$	$2N(1+\epsilon_3)$
Time	$T\frac{2I+S}{I}$	$T\frac{I+2S}{I}$	$T(1+\delta_1)$	$T(1+\delta_2)$	$T(1+\delta_3)$

Table 5.5: Comparison of single fault detection methods.

5.3.4 Analysis

As mentioned earlier, algorithmic methods are perhaps the most efficient means of fault detection but cannot be applied to general programs. This section will restrict itself to comparing skewed replicated computation streams and interleaved replicated computation streams with the hardware methods previously described.

For all practical purposes, RS and TTCDD require $k = N$, since if $k < N$ the probability that a transient fault will not be detected because the spares (or tokens) were in the wrong place is $1 - \frac{k}{N}$. Of course, any single *permanent* fault will be detected with $k = 1$, since the token or spare will eventually arrive at the faulty processing element.

Table 5.5 summarizes the methods when applied to the task of detecting any single transient fault. In this table, N, T, I, and S are the number of processors, amount of time, and number of cell program instructions and moving data streams used by the original code. All running times in Table 5.5 are in terms of the original running time T. The ϵ and δ terms represent overhead due to the special circuitry required by the hardware methods. The hardware methods are similar because they have been normalized for single fault detection. Shombert estimates that $\delta_1 \leq 0.33$ for the RS case, and such a figure seems reasonable for the other methods (it is likely that $\delta_1 \geq \delta_2 \geq \delta_3$). The value $\delta = 0.33$ is comparable to a 2IRCS program operating on four data streams, or two data and two fault streams, with $I = 12$.

As can be seen, 2IRCS and all the hardware method execution times are similar even when $\delta < 0.33$. 2SRCS is an appealing solution when extra processing elements are not readily available.

5.4 Conclusions

Permanent faults in the B-SYS chips were detected with a suite of test programs; the simplicity of the Systolic Shared Register design lead to a simple test plan. The B-SYS prototype board, hampered by its slow bus interface, can nevertheless run at 108 million operations per second. A more sophisticated board, perhaps the most important future task involving the Brown Systolic Array, could execute over 1 billion operations per second using only ten chips.

This section also introduced the method of software fault detection, which provides flexible fault coverage at a reasonable speed. In contrast hardware approaches, the interleaved replicated computation stream (IRCS) and skewed replicated computation stream (SRCS) methods do not require special hardware and do allow the user to vary the degree of fault detection. As seen by the use of automatic program transformations, software fault detection is a simple and versatile alternative to specialized algorithmic methods and complicated hardware solutions.

Chapter 6

A Framework for Systolic Programming

THE systolic paradigm is possibly the most efficient means of harnessing the vast power of massively parallel processors. However, systolic programming language development has not kept pace with systolic hardware development. As has been seen, most systolic programming languages draw tools from a bag of special tricks that are neither intuitive nor elegant. This chapter proposes the New Systolic Language as a general solution to the problem of systolic programming. Additionally, the systolic data stream aspect of the SSR architecture is explored. The New Systolic Language and its stream model of data flow overcome the difficulties and adapt the advantages of existing systolic programming methods.

6.1 Overview

The features of the New Systolic Language (NSL) draw heavily on the analysis of systolic programming languages and design systems presented in Section 2.3. Recall that none of these languages is entirely satisfactory for systolic co-processor programming. Many force the user to consider the implementation details of a specific systolic machine, while others force the user into a very abstract world, requiring the specification of abstract dependency vectors and mappings. Although both of these views are helpful at times, it is felt that a new methodology for programming systolic co-processors is required. To repeat the conclusions of the language analysis (page 35), a systolic co-processing language should, in the author's opinion, have the following characteristics:

1. The language should cleanly separate systolic cell programs and data flow directives. Shared asynchronous variables are to be avoided because multiple references can produce unpredictable results. Instead, cell programs should be specified as pure transfer functions between systolic inputs and systolic outputs.

2. Programs should be able to execute both host and co-processor array functions, preferably in the context of a conventional language.

3. The programmer should be able to declare systolic data streams as such using concise flow directives. Stream declaration and initialization should be accomplished apart from the cell program, dissociating cell operation from macroscopic data flow and enabling the reuse of standard data flows and cell operations.

4. The language should be independent of low-level co-processor features and functions, hiding array topology, processing element architecture, array size, and the method of systolic communication from the user. The programmer should not have to specify the physical names of queues, ports, registers, or bits.

The goal is to design a general-purpose systolic programming environment suitable for a wide variety of machines, from the Connection Machine, which can emulate meshes of arbitrary dimension, to the fixed linear topology of the Brown Systolic Array. The programmer should be able to quickly, concisely, and naturally specify the systolic cell functions and data flows of common systolic algorithms without any knowledge of the target architecture, apart from the fact that it is capable of supporting systolic operations over some broad range of topologies.

By the nature of this thesis, NSL's focus is strongly, though not entirely, on the B-SYS programmable systolic co-processor. As will become evident in the programming examples of the next chapter, low-level B-SYS code is not the easiest programming method for the Brown Systolic Array, in spite of its efficient implementation of systolic communication. Thus, this chapter considers more intuitive ways to program both B-SYS and programmable systolic arrays in general. In concert with this, the study of the SSR architecture is completed with a full explanation and exploration of systolic data stream programming, the fourth defining attribute of the SSR architecture. Although the Systolic Shared Register architecture can produce very efficient NSL implementations, it is not required by the NSL paradigm. The abstract idea of a systolic stream can be implemented on any parallel processor that can provide or simulate communication between processing elements over a regular network.

The New Systolic Language has been designed for generic systolic co-processing systems with the following traits:

- There is a host and a co-processor. Instructions and data flow from the host to the co-processor while results flow from the co-processor to the host.

- The host allocates all co-processor resources.

- All common arithmetic and logical operations are supported by the co-processor, though some may require multiple instructions.

It is believed that these criteria fully represent the interactions between a programmable systolic co-processor and its host computer.

As a consequence of being designed for systolic programming, the language does not support arbitrary programming of individual processing elements. Currently, algorithms that require more than one equivalence class of cell program (see Section 2.1.4) must define systolic streams to manually distinguish processing elements, as is done in

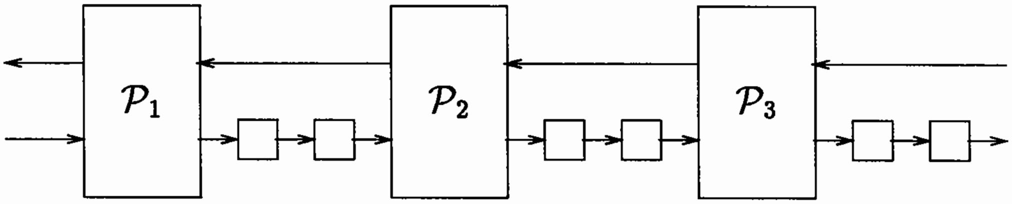


Figure 6.1: Two systolic streams.

the alternate transcription program of the next chapter (Section 7.1.4). It is expected that NSL will be extended to support any constant number of equivalence classes; for broadcast machines, the several logical instructions streams would be automatically converted to a single SIMD program, while for MIMD machines, all cell programs would be evaluated simultaneously.

The current NSL implementation is a prototype system which has proven useful for exploring the problem of systolic programming. As such, it generates code but does not execute it either on the B-SIM simulator or on the actual hardware (which is connected to a different machine). Nevertheless, experimentation with NSL has lead to many refinements and extensions to the original language specification. The NSL prototype has been designed with an eye toward implementing a complete system, and the prototype forms a firm basis for the development of an integrated system for systolic co-processor algorithm development, programming, simulation, animation, and execution.

This chapter continues with an overview of the New Systolic Language and a look at NSL cell programs and main programs. The C++ prototype implementation is then considered, followed by a review of possible enhancements to the prototype system. In conclusion, NSL is evaluated and future research problems in the domain of systolic programming languages are considered.

6.2 NSL Programs

The systolic stream is the most important aspect of the New Systolic Language. Conceptually, the systolic stream can be viewed as a flow of data buckets passing by the processing elements. During each time step, one bucket in the stream is directly accessible to each processing element. The value contained in the bucket may be used in computation and, if desired, changed. Before the next evaluation of the cell program, this bucket will have moved downstream to the next processing element or delay register, making room for a new input value.

Systolic speed refers to the number of time steps required for data to travel from one processing element to the next. That is, a stream implements some number of logical delay elements between processing elements which can be represented as shown in Figure 6.1. The figure diagrams a westward flowing stream of speed 1 and an eastward flowing stream of speed 3. Speed 0 streams are used for immobile, local data. Processing elements can also look small distances upstream and downstream, accessing future inputs

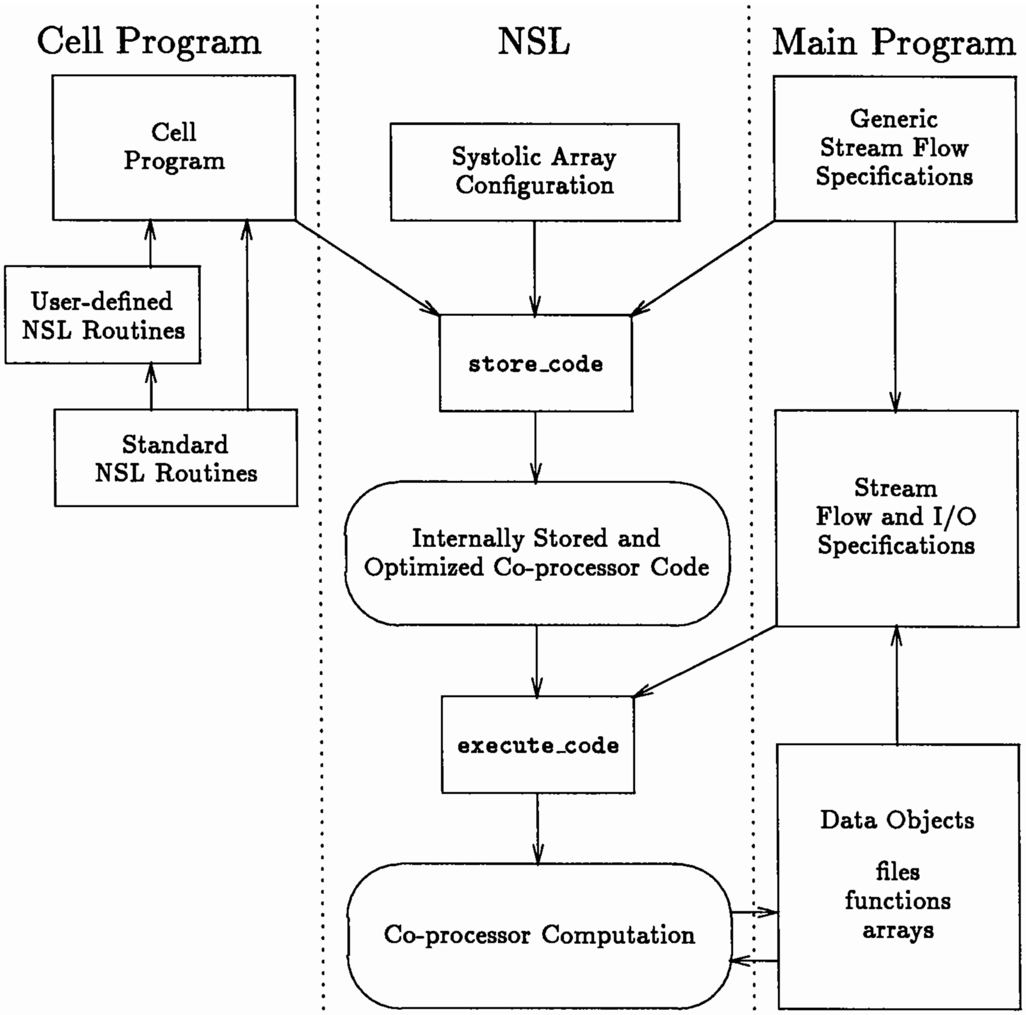

Cell Program
NSL
Main Program
Cell Program
Systolic Array Configuration
Generic Stream Flow Specifications
User-defined NSL Routines
store_code
Standard NSL Routines
Stream Flow and I/O Specifications
Internally Stored and Optimized Co-processor Code
execute_code
Data Objects
files
functions
arrays
Co-processor Computation

Figure 6.2: Overview of New Systolic Language programming.

and past outputs, as shall be described latter.

Systolic streams are defined in NSL main programs and, using the `store_code()` and `execute_code()` NSL procedures, are passed to the systolic cell programs. Systolic cell programs do not concern themselves with stream speed or type: NSL will assign registers and perform functions according to the declared type and speed of the systolic stream at runtime.

As seen in Figure 6.2, the major parts of the NSL system can be divided into three categories: the cell program, the main program, and the NSL system. As mentioned, the NSL cell program specifies computation on abstract systolic data streams, independent of the macroscopic data movement. Cell programs can call user-defined routines which process NSL data types (registers, flags, and data streams) as well as a large number of predefined operators and functions. The NSL prototype system was developed in the object-oriented C++ language which provided the ability to overload operators.[41] Thus, common operations have been defined for all basic systolic data types (i.e., when X and Y are systolic streams, the operation X+Y is evaluated by NSL to generate co-processor instructions).

NSL routines and NSL cell programs are not called like common C++ routines. References to an NSL cell program are restricted to calls from other NSL routines and NSL cell programs and to use as a parameter to the store_code() function of Figure 6.2. The NSL system itself will call the cell program and, as a consequence of the overloaded operators, generate code for the systolic co-processor.

NSL main programs control the flow of information through the systolic co-processor by defining systolic data streams. Systolic stream flow specifications include data type (integer or character) and precision information as well as the direction and speed of flow. In the current implementation, flow direction is restricted to eastward and westward, the limits of a linear systolic array; an NSL implementation for mesh systolic arrays would enable several more directions of flow.

In addition to flow information, streams include input and output specifications. Whenever a stream moves through the array, an input is needed and an output produced at the array boundaries. These inputs and outputs can be linked to files, functions, and arrays. By default, the input is zero and the output is ignored.

When the NSL system is initialized, information about the co-processor or simulator size, topology, and architectural features is defined. (Much of this information is accessible to the user for the construction of routines that must depend on hardware parameters.) This initialization informs NSL the type of systolic array required by the programmer's application — the NSL system must emulate that architecture if it differs from the actual hardware. A complete NSL system could, for example, automatically execute hexagonal-mesh programs on square- or triangular-mesh co-processors.

After initialization, a call to the store_code() procedure in the user's program will generate co-processor code. The cell program and stream flow specifications are combined and converted to the co-processor's native tongue. Hardware-specific resource optimization may also be performed.

The systolic co-processor is controlled by the execute_code() routine. This NSL function retrieves the previously stored code and sends it to the co-processor in whatever manner is appropriate to the machine. Using the I/O specifications of the systolic streams, the execute_code() routine also sends input to and stores output from the co-processor. As mentioned, the stream I/O specifications can link these inputs and outputs to files, functions, or arrays. In essence, the store_code() procedure can be regarded as NSL compilation while execute_code() performs NSL execution. Note, however, that these are done dynamically within a C++ program, so that NSL parameters and function definitions can remain indeterminate until the C++ program is executed on the host.

In addition to overloading operators, the NSL prototype system takes advantage of several other features from C++. First, C++ can convert between object types given the appropriate specifications, for example the register part of the flag and register result of an arithmetic operation can be automatically extracted. Second, C++ allows the

```
#include "nsl.h"
void
horner (SStream& C, SStream& X, SStream& Y)
{
  Y = Y + (C * X);
}
```

Figure 6.3: NSL cell program for Horner's method.

definition of object constructors and destructors. When registers or flags are declared, NSL constructors are called to allocate one of the systolic co-processor's storage locations. Similarly, to destroy registers or flags, destructors are automatically called to return the resource to the pool of available resources. Thus, the scope of register and flag allocation follows normal scoping conventions.

6.2.1 Cell Programs

NSL cell programs are C++ procedures that make use NSL's special systolic data objects. No information about data flow, data type, or array topology is present in the cell program.

An NSL cell program for Horner's method (Section 2.1.2) is shown in Figure 6.3.[a] Following the first requirement for systolic programming, this cell program is a pure transfer function that does not involve the macroscopic flow of data; no information about the systolic streams, apart from their formal parameter names, is available to the cell program. A stream name (`Y`) as an rvalue (on the right-hand side of an assignment statement) refers to the input value of that stream, while a stream name as an lvalue (left-hand side) will set the output value of that stream (corresponding to `Y'` in the SDEF systolic design system example of Figure 2.8 on page 29). Stream names which do not occur as lvalues (`C` and `X`) are given the identity transformation (`C=C` and `X=X`), as with SDEF.[b] For more complicated access to the streams, several other options will be detailed latter.

Cell programs can use most C++ arithmetic and logical operations and can declare registers or flags for local use, as is done in the NSL cell program for sorting (Figure 6.4). In addition to the predefined NSL operators and routines, users may program additional routines involving the NSL data types for use with cell programs, such as the following minimization routine:

[a] For those unfamiliar with C++, the notation `SStream& C` indicates that the parameter `C` is a reference to a `SStream` object, equivalent to the Pascal **var** declaration. Constant parameters can be specified with the `const` keyword.[41]

[b] The notation of the Hearts example (Figure 2.9 on page 30) can also be used, giving direct access to the stream input (`Y.in()`) and output (`Y.out()`) registers.

```
#include "nsl.h"
void
sort (SStream& Max, SStream& Min)
{
  Flag f;
  Max = select (f = (Min > Max), Min, Max);
  Min = select (f, Max, Min);
}
```

Figure 6.4: NSL cell program for sorting.

Field	Meaning
`Config::n`	Array size.
`Config::topo`	Array topology.
`Config::reg`	Registers per register bank.
`Config::flags`	Flags per functional unit.
`Config::edges`	Number of banks accessible to each functional unit.
`Config::banks`	Number of banks per functional unit.
`Config::word`	Length of word in bits.
`Config::zero_flag`	Pointer to constant zero flag.
`Config::one_flag`	Pointer to constant one flag.
`Config::mask_flag`	Pointer to mask flag.

Table 6.1: Static *Config* class members.

```
Register
min (const Register& r1, const Register& r2)
{
  return (select ((r1 < r2), r1, r2));
}
```

Cell programs and NSL routines can also make use of C++ looping constructs to create functions which depend on the target architecture, such as the following NSL routine to generate a test vector for bridge faults:

```
Register
bridge_test (void)
{
  Register result;
  for (int i = 0 ; i < Config::word; i+= 2) {
    result << 2; result++;
  }
  return (result);
}
```

The static *Config* class stores the systolic co-processor's configuration — size, topology, bits per word, pointers to constant flags, and the like, as tabulated in Table 6.1.

NSL cell programs only have systolic streams as arguments and never return a value. Data is passed to and from the main program (running on the host) through the systolic

```
#include "nsl.h"
main(void)
{
  int n=5;
    // Array length 5, 16 registers, 8 flags, two directions,
    // 1 register bank per PE, linear array, 8-bit word.
  Config::setup (n, 16, 8, 2, 1, LINEAR, 8);

    // Fixed stream of one-byte integer coefficients
  FixSStream Coeff (INT, NSL_BYTE1);

    // Two mobile streams
  SStream X (EAST, NSL_SPEED1, INT, NSL_BYTE1);
  SStream Y (EAST, NSL_SPEED1, INT, NSL_BYTE1);

    // Preprocess and store the cell program.
  store_code ("Horner", horner, &Coeff, &X, &Y);

    // Use a file for input of coefficients,
    // reversing them from the order they appear
    // in the file.
  Coeff.source ("file1", n, REVERSE, 0);
    // Read n X inputs from file2, then default to 0.
  X.source ("file2", n, IDENT, 0);
    // Place n Y outputs in file3 after waiting n
    // steps (Y inputs default to 0)
  Y.sink ("file3", n, IDENT, n);

    // Execute the complete cell program until the
    // Y stream has received n outputs.
  execute_code ("Horner", &Coeff, &X, &Y);
}
```

Figure 6.5: NSL main program for Horner's method.

data streams according to their input and output specifications. General NSL routines, such as min() and bridge_test(), can process and return any NSL or C++ data structure.

In summary, NSL cell programs are C++ routines that make use of the NSL *Register*, *Flag*, and *SStream* data types. The systolic cell programs contain no information about data movement, array configuration, or stream initialization and result extraction; cell programs can be easily reused with different data movements.

6.2.2 Main Programs

The NSL calling routines of Figures 6.5 and 6.6 have full control over the systolic array: they configure the array, maintain the data streams, and call systolic cell programs as

```
#include "nsl.h"
main(void)
{
  int n=470;
  Config::setup (n, 16, 8, 2, 1, LINEAR, 8);

  FixSStream Local (INT, NSL_BYTE1, 0);
  SStream Result (EAST, NSL_SPEED1, INT, NSL_BYTE1);

  store_code ("Sort", sort, &Local, &Result);

    // Take n numbers in as-is order, then default to 255
    // (the infinity value)
  Result.source ("unsorted", n, IDENT, 255);
    // Sink n output results to a file, but start output
    // collection 2n steps into execution.
  Result.sink ("sorted", n, IDENT, 2*n);

  execute_code ("Sort", &Local, &Result);
}
```

Figure 6.6: NSL main program for sorting.

needed. First, the array is initialized to the type (in this case, linear) and size desired. This is followed by several data stream definitions. Then, the code is preprocessed and stored, and inputs and outputs are linked to the systolic streams. Next, the systolic program is executed, and finally the streams are released for future use (in this case, at the completion of the main program).

Co-processor code is generated by calling the store_code() procedure with a symbolic name, the cell program name, and as many stream specifiers as are required. The store_code() function executes the systolic routine, in which operations involving NSL objects generate co-processor code but do not perform any computation on the systolic co-processor. The code is then analyzed to determine if weaving is necessary and, if so, the number of cycles required for the phased systolic program (see Section 3.4.2).

After determining weave conditions, code is regenerated using the revised stream flow specifications and then is compressed to eliminate temporary variables. Figures 6.7 and 6.8 show the verbose and final B-SYS code generated by the NSL system for the Horner's method cell and main programs (multiplication has been replaced by an exclusive-or operation to shorten the B-SYS assembly language output). In addition to the standard instruction fields described in Section 4.1.1, numeric stream identifiers are given, corresponding to the read and write fields of B-ASM assembly language (Section 4.1.1). Whenever a stream is accessed, input and output values are written to and read from the co-processor according to the stream definition. The sink() and source() member functions of the stream class specify the stream inputs and outputs as files, functions, arrays, or default values.

After confirming that the systolic stream specifiers are appropriate to the stored

```
; Initialization of flags
! Rfn 0xaa  W0  W0  W0  PGfn 0x0 0x0 F7 F7     Set mask flag
! Rfn 0xaa  W0  W0  W0  PGfn 0x0 0x0 F6 F6     Set one flag
! Rfn 0xaa  W0  W0  W0  PGfn 0x0 0x0 F5 F5     Clear zero flag
! Rfn 0x0   E14 E14 E14 PGfn 0xf 0x0 F0 F0     Set register
! Rfn 0xaa  E14 E14 W15 PGfn 0xf 0x0 F5 F5     IOStream= Word copy
; Load fixed stream into array.
! Rfn 0xaa  E15 E15 W15 PGfn 0xf 0x0 F5 F5 <0> IOStream= Word copy
! Rfn 0xaa  E15 E15 W15 PGfn 0xf 0x0 F5 F5 <0> IOStream= Word copy
! Rfn 0xaa  E15 E15 W15 PGfn 0xf 0x0 F5 F5 <0> IOStream= Word copy
! Rfn 0xaa  E15 E15 W15 PGfn 0xf 0x0 F5 F5 <0> IOStream= Word copy
! Rfn 0xaa  E15 E15 W15 PGfn 0xf 0x0 F5 F5 <0> IOStream= Word copy
! Rfn 0xaa  E15 E15 W15 PGfn 0xf 0x0 F5 F5 <0> IOStream= Word copy
; Execute_code routine Horner (0)
; Code for Horner cycle 0 of 1
! Rfn 0x66  W15 W14 W11 PGfn 0xf 0x0 F5 F5     Logical_bop XOR i=0
! Rfn 0x96  W11 W13 W12 PGfn 0x6 0x8 F5 F4     Arith_op ADD i=0
! Rfn 0xaa  W12 W12 E13 PGfn 0xf 0x0 F5 F5 <2> IOStream= Word copy
! Rfn 0xaa  W14 W14 E14 PGfn 0xf 0x0 F5 F5 <1> IOStream= Word copy
```

Figure 6.7: Uncompressed B-SYS code generated for Horner's method.

```
; Initialization of flags
! Rfn 0xaa  W0  W0  W0  PGfn 0x0 0x0 F7 F7     Set mask flag
! Rfn 0xaa  W0  W0  W0  PGfn 0x0 0x0 F6 F6     Set one flag
! Rfn 0xaa  W0  W0  W0  PGfn 0x0 0x0 F5 F5     Clear zero flag
! Rfn 0x0   E14 E14 E14 PGfn 0xf 0x0 F0 F0     Set register
; Load fixed stream into array.
! Rfn 0xaa  E15 E15 W15 PGfn 0xf 0x0 F5 F5 <0> IOStream= Word copy
! Rfn 0xaa  E15 E15 W15 PGfn 0xf 0x0 F5 F5 <0> IOStream= Word copy
! Rfn 0xaa  E15 E15 W15 PGfn 0xf 0x0 F5 F5 <0> IOStream= Word copy
! Rfn 0xaa  E15 E15 W15 PGfn 0xf 0x0 F5 F5 <0> IOStream= Word copy
! Rfn 0xaa  E15 E15 W15 PGfn 0xf 0x0 F5 F5 <0> IOStream= Word copy
! Rfn 0xaa  E15 E15 W15 PGfn 0xf 0x0 F5 F5 <0> IOStream= Word copy
; Execute_code routine Horner (0)
; Code for Horner cycle 0 of 1
! Rfn 0x66  W15 W14 W11 PGfn 0xf 0x0 F5 F5     Logical_bop XOR i=0
! Rfn 0x96  W11 W13 E13 PGfn 0x6 0x8 F5 F4 <2> Arith_op ADD i=0
! Rfn 0xaa  W14 W14 E14 PGfn 0xf 0x0 F5 F5 <1> IOStream= Word copy
```

Figure 6.8: B-SYS code generated for Horner's method.

code, the execute_code() routine locates code for the named routine and executes it on the co-processor (or, in the case of the NSL prototype system, prints out the code). The code is executed until all outputs have been generated, as determined by the stream specification. A step_code() function is available for running cell programs through one or more iterations instead of until output completion.

Two iterations of the NSL sorting program are shown in Figure 6.9. Results are

```
; Initialization of flags
! Rfn 0xaa  W0  W0  W0  PGfn 0x0 0x0 F7 F7     Set mask flag
! Rfn 0xaa  W0  W0  W0  PGfn 0x0 0x0 F6 F6     Set one flag
! Rfn 0xaa  W0  W0  W0  PGfn 0x0 0x0 F5 F5     Clear zero flag
! Rfn 0x0   E14 E14 E14 PGfn 0xf 0x0 F0 F0     Set register
; Execute_code routine Sort (0)
; Code for Sort cycle 0 of 2
! Rfn 0x96  W14 W15 E12 PGfn 0x9 0x4 F5 F4     Rel_op SUB
! Rfn 0xac  W15 W14 E13 PGfn 0xf 0x0 F4 F4 <1> select
! Rfn 0xac  W14 W15 W15 PGfn 0xf 0x0 F4 F4 <0> select
; Code for Sort cycle 1 of 2
! Rfn 0x96  W13 W15 W12 PGfn 0x9 0x4 F5 F4     Rel_op SUB
! Rfn 0xac  W15 W13 E14 PGfn 0xf 0x0 F4 F4 <1> select
! Rfn 0xac  W13 W15 W15 PGfn 0xf 0x0 F4 F4 <0> select
```

Figure 6.9: B-SYS code generated for sorting.

automatically woven between the registers E_{13} and E_{14}. As can be seen, this assembly language code is semantically identical to the B-SIM sorting program of Figure 4.4 on page 54, though of course the NSL source code is much easier to understand.

6.3 The NSL Implementation

The New Systolic Language prototype is a collection of C++ objects which allocates and deallocates co-processor resources. The objects, corresponding to flags, sets of registers, and systolic streams, can be manipulated with a wide variety of predefined operators and functions. In addition, there are several functions available for controlling the co-processor (for example, setting and clearing the mask bit) and the available resources (for example, accessing all data in a systolic stream, not just the cell inputs and outputs).

The structure of the basic data objects (classes *Register*, *Flag*, and *RFpair*) is displayed in Figure 6.10. The *Register* object is an abstract register: it has its own type (i.e., signed integer, unsigned integer, or character) and precision (in bytes). Each physical register (in the case of B-SYS, sixteen) and flag (in the case of B-SYS, eight) has its own unique identifying object, a *Reg_obj* or *Flag_obj*. These are allocated and deallocated by the *Register* and *Flag* classes according to normal C++ scoping rules. A reference counter is used to determine when an object may be safely returned to the pool of free resources. Only pointers are used to refer to these basic resources because of C++'s proclivity to copy objects as needed. Thus, the reference counter and pointer use ensures both that multiple copies of a resource do not exist and that resources do not vanish until all references to them have been eliminated. In the case of B-SYS, three of the flags are permanently removed from the resource pool during configuration for special use, and their addresses are stored in the static *Config* class, as can be seen from Table 6.1.

The *RFpair*, consisting of a *Flag* and a *Register* (either can have zero length), is the basic unit of NSL computation. Most C++ operators have been defined for *RFpair*

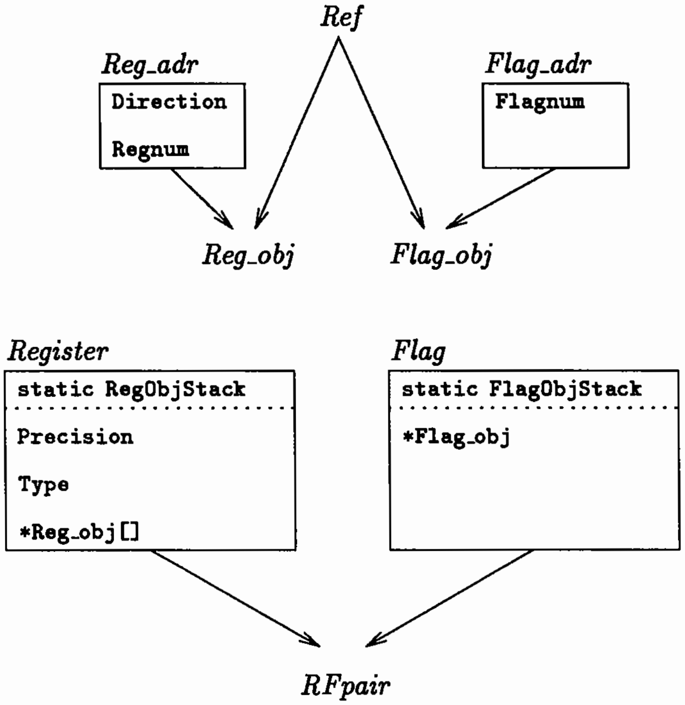


Figure 6.10: NSL prototype data objects.

objects, as well as methods for automatically converting *Register*, *Flag*, and *SStream* objects to the *RFpair* class. *RFpair* objects can also be manually or automatically cast to *Register* or *Flag* objects, in the process of which the other half of the *RFpair* becomes inaccessible. The operators and functions implemented for the NSL prototype system are displayed in Table 6.2. Unfortunately, the C++ selection operator :? cannot be overloaded.

Register and *Flag* objects do not persist between cell program iterations: systolic streams must be used for persistent data, both flowing and fixed. Each stream acquires some number of registers according to its data type, speed, and use of weaving. Data in a speed *n* stream must take *n* steps to reach the next processing element, so *n* delay registers must be allocated to the stream. A stream moving at a speed of one requires one register, while a stream moving at a speed of two requires one buffer element and one other register. One additional register is allocated when weaving is required. For example, the Max stream in the sorting example uses two registers since it is a speed 1 stream with weaving.

SStream objects (Figure 6.11), as seen in the code examples, must also specify inputs and outputs for the stream, though a lack of either is acceptable. The structure of the *Inputobj* and *Outputobj* classes used by *SStream* objects is displayed in Figure 6.12. Each member stores a mapping, such as as-is or reverse order (similar to Hearts), a number of inputs to give or outputs to receive, and the number already processed. The various

Operator	Meaning	Operator	Meaning
-	Unary minus	~	Logical negation
&	Bitwise and	&=	Bitwise and
^	Bitwise exclusive-or	^=	Bitwise exclusive-or
|	Bitwise or	|=	Bitwise or
<<	Left shift	<<=	Left shift
>>	Right shift	>>=	Right shift
<	Less than	<=	Less than or equal
>	Greater than	>-	Greater than or equal
==	Equal	!=	Not equal
+	Addition	+=	Addition
-	Subtraction	-=	Subtraction
*	Multiplication	*=	Multiplication
++	Increment	--	Decrement
&&	Flag and	||	Flag or
!	Flag negation	`select()`	Selection ?:
`match()`	Matching bit position	`nomatch()`	No matching bit position
`force_mask()`	Unconditionally set mask	`use_mask()`	Do or do not obey mask flag

Table 6.2: NSL operators and functions.

subclasses of *IOobj* store additional information, such as default input values and delays before recording output values, buffers for files, and pointers to arrays or functions.

An NSL program for sequence comparison (Sections 2.1.3 and 7.1) is shown in Figure 6.13. This program illustrates the use of functions and arrays for input and output: the function `iweight()` returns the value $d_{t,0} = t$ for each time step t (the index of the time step is passed to the routine when called by NSL). More importantly, the program illustrates advanced manipulation of the systolic stream object type. In the sequence comparison algorithm, three previous results are required: $d_{i-1,j-1}$, $d_{i,j-1}$, and $d_{i-1,j}$, the first two having been computed in $\mathcal{F}_{j-1}$ (at times $t-2$ and $t-1$) and the remaining in $\mathcal{F}_j$ (at time $t-1$). Since cost data is required from two time steps ago, the cost stream moves at a speed of 2 (thus, three registers are allocated to the stream because weaving is required). The output of $\mathcal{F}_j$'s cell program will be $d_{i,j}$, and the input is $d_{i-1,j-1}$. The problem is to access $d_{i,j-1}$, which will be *next* time step's stream input, and $d_{i-1,j}$, which was *last* time step's stream output. In short, $\mathcal{F}_j$ must look upstream and downstream from its current position. In NSL, these values are accessible using array notation, indexing the systolic data stream relative to the current input or output. Thus, the expression `Cost[0]` is equivalent to both `Cost.in()` and `Cost` as an rvalue, all of which refer to the stream input value $d_{i-1,j-1}$. `Cost[1]` retrieves the value one time unit down the systolic stream, or $d_{i-1,j}$ (produced by $\mathcal{F}_j$ last time step). Similarly, negative indices retrieve values upstream, so `Cost[-1]` retrieves the value one time unit up the systolic stream, or $d_{i,j-1}$ (produced by $\mathcal{F}_{j-1}$ last time step). Thus, functional unit $\mathcal{F}_j$ sees `Cost` as a stream of data flowing past it, and is able to look short distances upstream and downstream as it computes values to place in that stream. The implementation of these options is simple because shared registers are used for systolic communication. The methods available for

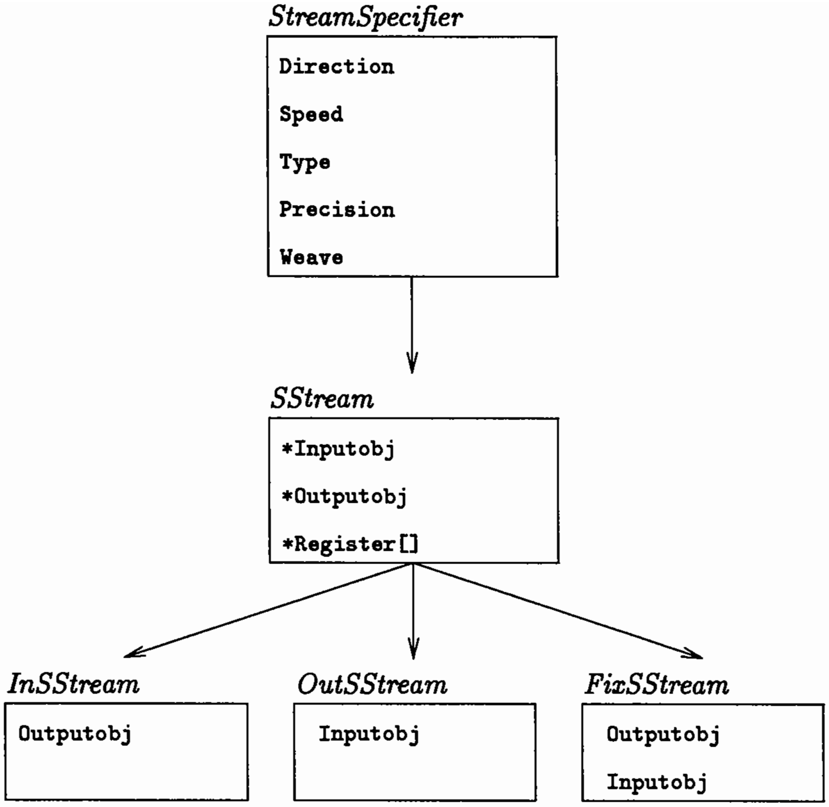


Figure 6.11: NSL prototype systolic stream objects.

use with *SStream* objects are tabulated in Table 6.3.

The resulting B-SYS code for the sequence comparison problem is displayed in Figure 6.14. As can be seen, the current NSL compression algorithm is not optimal: the cell program's length would be reduced by one instruction (17%) if the character matching and character movement (the first and last statements) were combined. The stream of costs has been allocated three registers (W_{11}, W_{12}, and W_{13}) because it is a speed 2 stream with weaving.

6.4 Implementation Enhancements

The current NSL prototype implementation fully understands all the features and operators mentioned herein, but although it generates valid B-SYS code it is unable to run the code on either the simulator or the co-processor hardware. Since the goal of this research was to investigate languages for general-purpose systolic programming, an examination of several NSL applications and the generated B-SYS code proved sufficient to evaluate the programming paradigm.

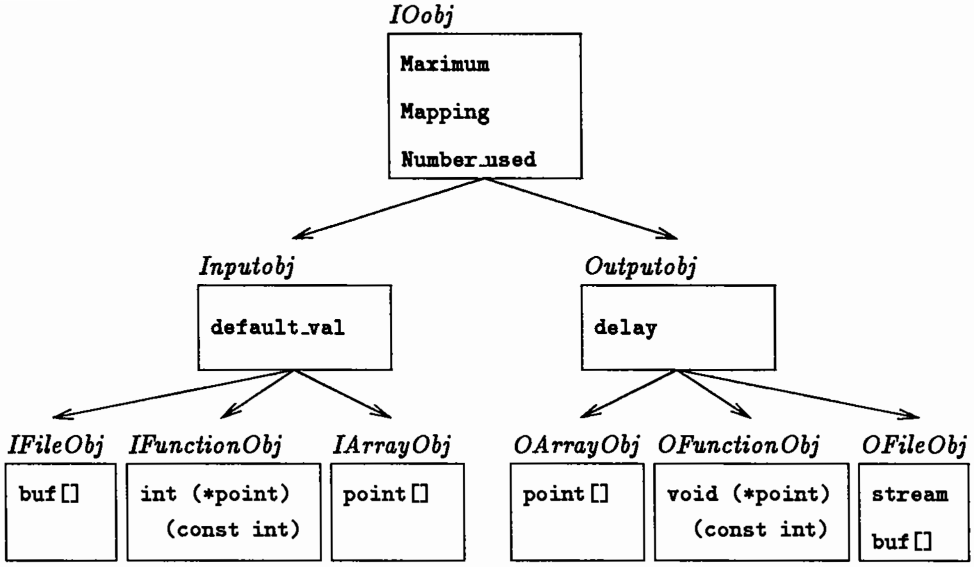


Figure 6.12: NSL prototype I/O objects.

Experimentation with the New Systolic Language prototype system has lead to many observations about and refinements of the New Systolic Language. Several of these have already been incorporated in the NSL system, in particular the ability to access upstream and downstream data just described. However, there are several implementation enhancements which should be incorporated into future versions of the NSL system. These include:

1. Systolic streams should be modified to have three principle flow attributes: direction or location, speed, and persistence. In the current implementation, only neighboring locations, such as east or west, may be specified to indicate logically adjacent processing elements. This could be generalized to allow connections to any relatively-addressed local processing element. For example, in one of the applications to be discussed in the next chapter (Section 7.1.5), it would be useful to specify a stream that automatically hops over processing elements, sending data from each $\mathcal{P}_i$ to $\mathcal{P}_{i+2}$ instead of the standard $\mathcal{P}_{i+1}$.

   The speed of a systolic stream can remain as defined in the NSL prototype system, equal to the number of delay elements between processing elements. However, in conjunction with the systolic speed, a persistence attribute should be added to each stream to indicate how long data persists in the systolic stream beyond the minimum requirements of the speed definition. Thus, a fixed systolic stream (speed 0) with a persistence of 2 would make available both the current step's and the previous step's stream input value. Similarly, mobile streams should allow persistences in case stream input values must be reused latter.

```
#include "nsl.h"
#define DEL_COST 1
void
sequence (SStream& Char1, SStream &Char2, SStream &Cost)
{
  // Cost input is d_{i-1,j-1}
  // Cost[-1] is last step's input,  or d_{i,j-1}
  // Cost[+1] is last step's output, or d_{i-1,j}.
  Cost = select (match (Char1, Char2),
                 Cost,
                 select (Cost[-1] < Cost[+1],
                         Cost[-1], Cost[+1]) + DEL_COST);
}
int
iweight (const int n)  // n-th initial weight d_{n,0}
{
  return n;
}
main(void)
{
  int n = 5;
  int final_cost;
  Config::setup (n, 16, 8, 2, 1, LINEAR, 8);

    // Fixed stream of one-byte characters.
  FixSStream Seq1 (CHAR, NSL_BYTE1);

    // Two mobile streams
  SStream Seq2   (EAST, NSL_SPEED1, CHAR, NSL_BYTE1);
  SStream Weight (EAST, NSL_SPEED2, INT, NSL_BYTE1);

  store_code ("Sequence", sequence, &Seq1, &Seq2, &Weight);

  Seq1.source ("seq1", n);
  Seq2.source ("seq2", n);
    // Use a function for n input values.
  Weight.source (iweight, n);
    // Only save one result (in an array).
  Weight.sink (&final_cost, 1, IDENT, 2*n-1);

  execute_code ("Sequence", &Seq1, &Seq2, &Weight);

  cout << "Sequence Distance: " << final_cost;
}
```

Figure 6.13: NSL cell program for sequence comparison.

Function	Meaning
S	Stream input (rvalue) or output (lvalue).
S.in()	Stream input value.
S[0]	Stream input value.
S[-i]	Upstream input value.
S.out()	Stream output value.
S[+i]	Downstream output value.
S=	Assignment of stream output value.
S.source()	Assignment of file, function, or array as stream's input source.
S.sink()	Assignment of file, function, or array as stream's result sink.
S.force_weave()	Force S to be woven.

Table 6.3: *SStream* member functions.

```
; Initialization of flags
! Rfn 0xaa  W0  W0  W0  PGfn 0x0 0x0 F7 F7     Set mask flag
! Rfn 0xaa  W0  W0  W0  PGfn 0x0 0x0 F6 F6     Set one flag
! Rfn 0xaa  W0  W0  W0  PGfn 0x0 0x0 F5 F5     Clear zero flag
! Rfn 0x0   E14 E14 E14 PGfn 0xf 0x0 F0 F0     Set register
; Load fixed stream into array.
! Rfn 0xaa  E15 E15 W15 PGfn 0xf 0x0 F5 F5 <0> IOStream= Word copy
! Rfn 0xaa  E15 E15 W15 PGfn 0xf 0x0 F5 F5 <0> IOStream= Word copy
! Rfn 0xaa  E15 E15 W15 PGfn 0xf 0x0 F5 F5 <0> IOStream= Word copy
! Rfn 0xaa  E15 E15 W15 PGfn 0xf 0x0 F5 F5 <0> IOStream= Word copy
! Rfn 0xaa  E15 E15 W15 PGfn 0xf 0x0 F5 F5 <0> IOStream= Word copy
! Rfn 0xaa  E15 E15 W15 PGfn 0xf 0x0 F5 F5 <0> IOStream= Word copy
; Execute_code routine Sequence (0)
; Code for Sequence cycle 0 of 2
! Rfn 0xaa  W15 W14 E10 PGfn 0xf 0x8 F5 F4     Rel_op MATCH
! Rfn 0x96  W13 E12 W10 PGfn 0x9 0x4 F5 F3     Rel_op SUB
! Rfn 0xac  W13 E12 E10 PGfn 0xf 0x0 F3 F3     select
! Rfn 0x5a  E10 E10 E10 PGfn 0xa 0x0 F6 F3     Arith_uop INCR
! Rfn 0xac  W12 E10 E12 PGfn 0xf 0x0 F4 F4 <2> select
! Rfn 0xaa  W14 W14 E14 PGfn 0xf 0x0 F5 F5 <1> IOStream= Word copy
; Code for Sequence cycle 1 of 2
! Rfn 0xaa  W15 W14 W11 PGfn 0xf 0x8 F5 F4     Rel_op MATCH
! Rfn 0x96  W12 E13 E11 PGfn 0x9 0x4 F5 F3     Rel_op SUB
! Rfn 0xac  W12 E13 W11 PGfn 0xf 0x0 F3 F3     select
! Rfn 0x5a  W11 W11 W11 PGfn 0xa 0x0 F6 F3     Arith_uop INCR
! Rfn 0xac  W13 W11 E13 PGfn 0xf 0x0 F4 F4 <2> select
! Rfn 0xaa  W14 W14 E14 PGfn 0xf 0x0 F5 F5 <1> IOStream= Word copy
```

Figure 6.14: B-SYS code generated for sequence comparison.

2. Support for on-board program storage should be added. The NSL system would store a selection of routines on the co-processor board for autonomous execution, perhaps using one of the many well-studied instruction caching algorithms. The separation of the `store_code()` and `execute_code()` routines will aid in this task. Additionally, the optimization routines used by NSL should be improved to more efficiently allocate the co-processor's resources. This would include, for example, examination of cell programs for integer constants which can then be loaded before the execution of a cell program (currently, integer constants are constructed in every processing element during each cell program iteration). More efficient means of accessing the mask flag should also be considered.

3. When insufficient memory is available in processing elements for execution of the cell program, NSL should automatically allocate multiple processing elements per systolic cell program, reconfiguring the systolic streams as necessary. The protein scanning algorithm of Section 7.1.5 will illustrate the need of this feature.

4. Support for programmable systolic co-processors other than the Brown Systolic Array should be included, as well as a general-purpose simulator interface, possibly based on the Tango system mentioned in Section 4.1.2.

As can be seen, the NSL framework is not yet entirely full, however experimentation with the prototype system has lead to many important ideas and methods for systolic programming.

6.5 Conclusions

The New Systolic Language greatly expedites B-SYS programming, freeing the programmer from many tedious tasks. The obvious question is whether or not it has satisfied the goal of providing a simple systolic programming language for the creation of machine-independent programs using a logical structure suitable for systolic co-processors. As the examples of this chapter have illustrated, NSL does provide a concise and intuitive interface for systolic co-processor programming. NSL uses the clean separation of cell function and data movement present in SDEF, while remaining suitable for use with real hardware. The pitfalls of shared asynchronous variables and low-level systolic communication directives have been deftly sidestepped. The design is not limited to any particular machine or network topology, and future extensions could make it truly independent of hardware.

The most obvious future development issue is the extension of NSL to topologies of two or more dimensions. In concert with this, stream direction specifications to access arbitrary processing elements within these topologies should be added. A vector notation similar to that of the SDEF system could specify, for example, a stream flowing to the processing element two units north and one unit east. More common flows, such as northwest, would only require simple textual specification. Depending on actual network topology, some of these streams may require the generation of extra co-processor data movement instructions (i.e., simulating an octagonal mesh on a square mesh). Also, support for two-dimensional input and output specifications, perhaps similar to those of Hearts, should be provided: a systolic stream composed of entire rows from a matrix, distributed along a column of processing elements, should be simple to specify and easy to use. Many of these issues should be developed and refined concurrently with the stream enhancements discussed in the previous section.

Support for wavefront programming should be investigated. Some algorithms, such as systolic data compression, do not have a fixed relationship between input to and output from the array: the delay between input and output depends on how compactly the text can be compressed. Methods for automatically providing sentinels or other mechanisms which implement wavefront programming should be evaluated.

Since NSL is intended for arbitrary systolic co-processors without program modification (apart from NSL initialization via the *Configuration* call), support for oversized and undersized arrays must be developed. In the former case, this could involve controlling a variable-length co-processor, masking certain processing elements, or user-guided cell program analysis to find algorithmic methods of coping with large arrays. For example, in the case of sequence comparison, extra processing elements can be loaded with wildcard characters which match all other characters. Thus, by the sequence comparison algorithm, these processing elements will not affect the cost values exiting the array. Similarly, padding strings with null characters (ϕ) will have an entirely predictable effect on the distance computation. Performing functions on undersized arrays is a more complicated, though still achievable, goal. With guidance from the programmer, NSL should be able to partition problems for undersized arrays by storing intermediate results in the host's memory.

The New Systolic Language prototype system is a framework for a complete and integrated systolic co-processing environment for arbitrary topologies and implementations. Cell programs are, as they should be, independent of both the implementation and the form of systolic communication; cell programs do not access specific data queues or communication registers, but only access the abstract systolic stream data type. NSL main programs, written in C++, can execute host and co-processor functions in a simple and intuitive manner. Additionally, NSL has been designed for use with independently controlled co-processors; a complete NSL system would extract the highest possible performance from systems similar to B-SYS*.

Systolic communication is expressed cleanly and concisely with the definition of systolic stream objects. The user can link files, functions, and arrays to the streams, and additional input and output specifications can be easily added to the NSL class library. Finally, the user can specify different topologies, though at present only linear systolic arrays are supported.

The case for the New Systolic Language is further supported in the next chapter, wherein several applications are programmed in NSL.

Chapter 7

B-SYS Applications

IN an effort to gauge the performance and success of the Brown Systolic Array, this chapter considers the programming of several applications on the system. The problem which has had the greatest influence on the design of B-SYS has been sequence comparison, a problem critical to the analysis of the human genome. A system such as B-SYS* (Section 5.2.4) would bring both high performance and programmability to this problem, features which are exploited in the several sequence comparison variations discussed herein. The Brown Systolic Array is not just a programmable sequence comparison machine, it can perform a variety of other tasks to speed combinatorial computations. As an example, a B-SYS program for transitive closure, traditionally solved on 2-dimensional systolic arrays, is examined.

7.1 Sequence Comparison

As has been mentioned, there are many variations on the basic sequence comparison problem, including use of different character matching costs, searches for a subsequence, searches for the longest common subsequence, and the introduction of gap penalties for breaking or joining a sequence.[135] P-NAC can only perform sequence comparison over a four-character alphabet with hardwired costs.[110] BioSCAN, first mentioned in the footnote on page 18, can perform subsequence searches over a 28-character alphabet with programmable costs but without insertion or deletion operations (only point mutations are allowed), as described in Section 7.1.5.[35] Splash can theoretically perform many types of sequence comparison, however more complicated versions, such as comparisons over large alphabets or with different cost metrics, will result in a low number of systolic cells per field-programmable gate array.[58] Because it provides a high concentration of fully programmable processing elements, the Brown Systolic Array is well suited to be a general-purpose programmable sequence comparison engine.

To begin this examination of sequence comparison on the Brown Systolic Array, two sequence comparison programs for the algorithm considered in Sections 2.1.3, 5.2.3, and 6.3 are analyzed. Next, a sequence comparison algorithm with an affine (or gap) cost function is presented. These three algorithms are most appropriate for nucleic-acid

x	y	$x-y$	$x-y$ mod 256	Z	R	Smaller
6	4	2	2	0	00000010	y
4	6	−2	−2	1	11111110	x
255	2	253	−3	0	11111101	x
2	255	−253	3	1	00000011	y

Table 7.1: Differences modulo 256.

sequence comparison, in which all four characters (nucleotides) are treated identically. Next, the problem of finding an amino acid encoding region within a nucleic acid is examined. Finally, continuing to move up the hierarchy of biological structures, the analysis of two amino acid chains (proteins) in a manner similar to that of BioSCAN is considered.

7.1.1 Simple Sequence Comparison

There is a problem with the sequence comparison algorithm of Figures 6.13 on page 108. If the sequences are over 128 characters long, the total edit distance could exceed 256, a number beyond the range of one-byte integers. That is, if the two sequences have no matching characters (for example, one being a sequence of the letter 'A' and the other of the letter 'C'), the final cost $d_{128,128}$, corresponding to deleting the entire first string (at a cost of 128) and inserting the entire second string (128), will be 256. With one-byte integers, the nefarious principle of integer wraparound will turn this result into $d_{128,128} = 0$, incorrectly indicating a perfect match.

One solution is the use higher-precision integers, however this will make the cell program depend on the length of the array: longer arrays and sequences will require the assignment of more registers to the cost data stream to maintain adequate precision. As an alternative, BioSCAN imposes minimum (0) and maximum (16 384) values on $d_{i,j}$, the latter being treated as an infinity value which is never decremented.[142] Character matching costs must be scaled according to sequence length because the threshold is not programmable. Regardless of scaling, these arbitrary caps will limit BioSCAN's effectiveness when comparing sequences of fifty thousand or more characters.

A much better solution is the modulo sequence comparison method developed by Lopresti, who noticed that although global cost differences between two strings can be large, local differences (such as the difference between $d_{i-1,j}$ and $d_{i,j-1}$) are always small.[109] The problem is to determine which of two values ($d_{i-1,j}$ or $d_{i,j-1}$) is the smaller given that integer wraparound may have occurred. Thus, when comparing values modulo 256, the number 255 is *smaller* than the number 1 because $1-255 = 2 \pmod{256}$, and the alternative (a difference of 254) is impossible by the magnitude constraint on local differences. The four cases in comparing two numbers x and y modulo 256 are shown in Table 7.1, along with the Z flag and R byte results of each subtraction. As can be seen, it is not the carry output Z that indicates which value is the smaller modulo 256,

```
#include "nsl.h"
#define DEL_COST 1
void
sequence (SStream& Char1, SStream& Char2, SStream& Cost)
{
  Flag f;

    // set f to bit 7 of the difference
  f = (Cost[-1] - Cost[+1]) << 1;

  Cost = select (match (Char1, Char2),
                 Cost,
                 select (f, Cost[-1], Cost[+1]) + DEL_COST);
}
```

Figure 7.1: NSL modulo sequence comparison program.

but the most significant bit of the result. Modulo m sequence comparison can be used whenever the local differences are strictly less than $\frac{m}{2}$, 128 for 8-bit integers (that is, when $|a-b| < \frac{m}{2}$, $\min(a, b)$ can be unambiguously determined by comparing $a \pmod m$ and $b \pmod m$)).

An NSL cell program for modulo sequence comparison is displayed in Figure 7.1. A C program similar to that of Figure 7.2 was used to drive the B-SYS prototype system as the board was being assembled.[a] In this program, the `ZInstruction()` and `Instruction()` macros execute an instruction on the B-SYS prototype board with a zero input or an arbitrary input, respectively. The `ReadBsys()` macro queries the prototype board for an output value. The macros follow from the port assignments of Table 5.3 on page 81; the macros are defined in Figure 7.3. As can be seen, input and output data values are passed to and from the prototype board in complemented form. The `post_bsys()` routine processes the modulo sequence comparison data to extract the true $d_{n,n}$ result by applying the constraint on local differences.

As mentioned in Section 5.2.3, this sequence comparison program was executed as the array was being built, and its timings are tabulated in Table 5.4 on page 82. As can be seen in Figure 7.4, the Brown Systolic Array prototype system greatly outperforms the host 25 MHz Intel 80386 microprocessor.

7.1.2 Comparison of One String Against Many

Unfortunately, the sequence comparison program of the previous example only maintains 50% processor utilization: as the string enters the array all functional units to the east of the first character are not performing useful computation, and as the string exits the

[a] In the real version, the $2n$ loop iterations are split into two halves. During the first part, the input sequence s is streamed into the array but results are not collected. In the second part, null characters are used for character input while cost results are stored. As commented in the figure, not shown is the initialization during which the sequence t is loaded into the array and several flags and registers are set.

```
int bcompare  (register unsigned char *s,  /* moving string */
               register unsigned char *t,  /* fixed string  */
               int n,                      /* array size    */
               register unsigned char *r)  /* result array  */
{
  int i;

  /* Store sequence t in the array and initialize   */
  /* flags and registers.  F7 = 0, F6 = 1.          */
  /* E0 = fixed character, W1 = moving character    */

  /* First n steps (loading sequence) and second n steps    */
  /* combined for simplicity                                */
  for (i = 0, _CX=0 ; i < 2*n ; i+=2) {
      /* Compare two previous costs (W2, E2) */
    ZInstruction(1, xorABC,      W2, E2, WF, Zsub,    F7, F1);
      /* But do the comparison mod 256       */
    ZInstruction(1, fnA,         WF, WF, WF, Zmsb,    F1, F1);
      /* Select the minimum, place in WF     */
    ZInstruction(1, selectABonC, W2, E2, WF, Zconst,  F1, F1);
      /* Add 1, the insertion or deletion cost, to WF        */
    ZInstruction(1, xorAC,       WF, WF, WF, Zadda,   F6, F1);
      /* Compare the characters in W1 and E0, passing the W1 */
      /* character east and feeding a character to the array */
    Instruction (1, fnA,         W1, E0, E1, matchAB, F7, F2, *s);
      /* If the characters matched, use W4 as the cost,      */
      /* otherwise use WF. Store result in E4 and load d(t,0)*/
    Instruction (1, selectABonC, W4, WF, E4, Zconst,  F2, F2,_CX);
    ReadBsys (*r);
    _CX++; s++; r++;

      /* Repeat, exchanging the meanings of W2/E2 with W4/E4 */
    ZInstruction(1, xorABC,      W4, E4, WF, Zsub,    F7, F1);
    ZInstruction(1, fnA,         WF, WF, WF, Zmsb,    F1, F1);
    ZInstruction(1, selectABonC, W4, E4, WF, Zconst,  F1, F1);
    ZInstruction(1, xorAC,       WF, WF, WF, Zadda,   F6, F1);
    Instruction (1, fnA,         W1, E0, E1, matchAB, F7, F2, *s);
    Instruction (1, selectABonC, W2, WF, E2, Zconst,  F2, F2,_CX);
    ReadBsys (*r);
    _CX++; s++; r++;
  }
  return (post_bsys (r, n));
}
```

Figure 7.2: B-SYS modulo sequence comparison program.

```
#define W1_PORT  0x300
#define W2_PORT  0x304
#define W3_PORT  0x308
#define IN_PORT  0x30C
#define oport(port) asm {mov dx,port; out dx,ax; }
#define Instruction(Call,Rfn,A,B,R,GPfn,C,Z,inval) {            \
  _AX = ((GPfn) << 8) | (Rfn); oport (W1_PORT);                 \
  _AX = ((((R) << 5) | (B)) << 5) | (A)) << 1; oport (W2_PORT); \
  _AL = inval;  _AL = ~_AL;                                      \
  _AH = (((Call) << 3) | (Z)) << 4) | (C);                       \
  oport (W3_PORT); asm {jmp $+2;};}
#define ZInstruction(Call,Rfn,A,B,R,GPfn,C,Z) {                  \
  _AX = ((GPfn) << 8) | (Rfn); oport (W1_PORT);                 \
  _AX = ((((R) << 5) | (B)) << 5) | (A)) << 1; oport (W2_PORT); \
  _AX = (((((Call) << 3) | (Z)) << 4) | (C)) << 8) | 0xff;      \
  oport (W3_PORT); asm {jmp $+2;};}
#define ReadBsys(var)                                            \
  {asm {mov dx, IN_PORT; in ax,dx;} var = ~_AL;}
```

Figure 7.3: B-SYS prototype macro definitions.

array all functional units to the west of the final character are not performing useful computation. Database search typically involves comparing one string against many other strings (the database) to find interesting or similar sequences, and for this variation of sequence comparison close to 100% processor utilization can be maintained

To maintain high processor utilization, the systolic period (the time between streaming one sequence and the next through the array) must be reduced to approximately n from $2n$. To achieve this another systolic stream, a reset stream, is introduced to the algorithm. This stream will always have a value of either $\mathtt{00}_{16}$ or $\mathtt{FF}_{16}$. The $\mathtt{FF}_{16}$ value indicates that a reset should be processed. That is, a reset signal indicates that valid data (data involving the new sequence) is only available upstream from the current functional unit (downstream data involves the *exiting* sequence). In NSL, this reset stream can be used as shown in Figure 7.5. The reset signal forces a functional unit to choose Cost[-1] ($d_{i,j-1}$) as the minimum of $d_{i,j-1}$ and $d_{i-1,j}$, regardless of the true modulo minimum. The next problem is to ensure that this value is used in the computation of $d_{i,j}$. To prevent the old $d_{i-1,j-1}$ from being chosen as $d_{i,j}$ (as it is when the characters in $\mathcal{F}_j$ match), the null (non-matching) character is passed in the mobile character stream in conjunction with the reset signal. Finally, whenever the reset signal is introduced, the initial weights ($d_{i,0}$ values) sent to the array are restarted from 0.

The actual program used to test this on the prototype hardware was written in C, with a slightly more efficient reset computation (Figure 7.6). The generate and propagate control signals of $\mathtt{A}_{16}$ and $\mathtt{F}_{16}$ always propagate a carry input $C = 1$ to the carry output Z, and will also assert Z whenever the most significant bit of the reset stream is set. That is, when the reset stream input is $\mathtt{FF}_{16}$, a carry is always generated; without the reset signal, the flag indicating the modulo minimum of $d_{i,j-1}$ and $d_{i-1,j}$ is left

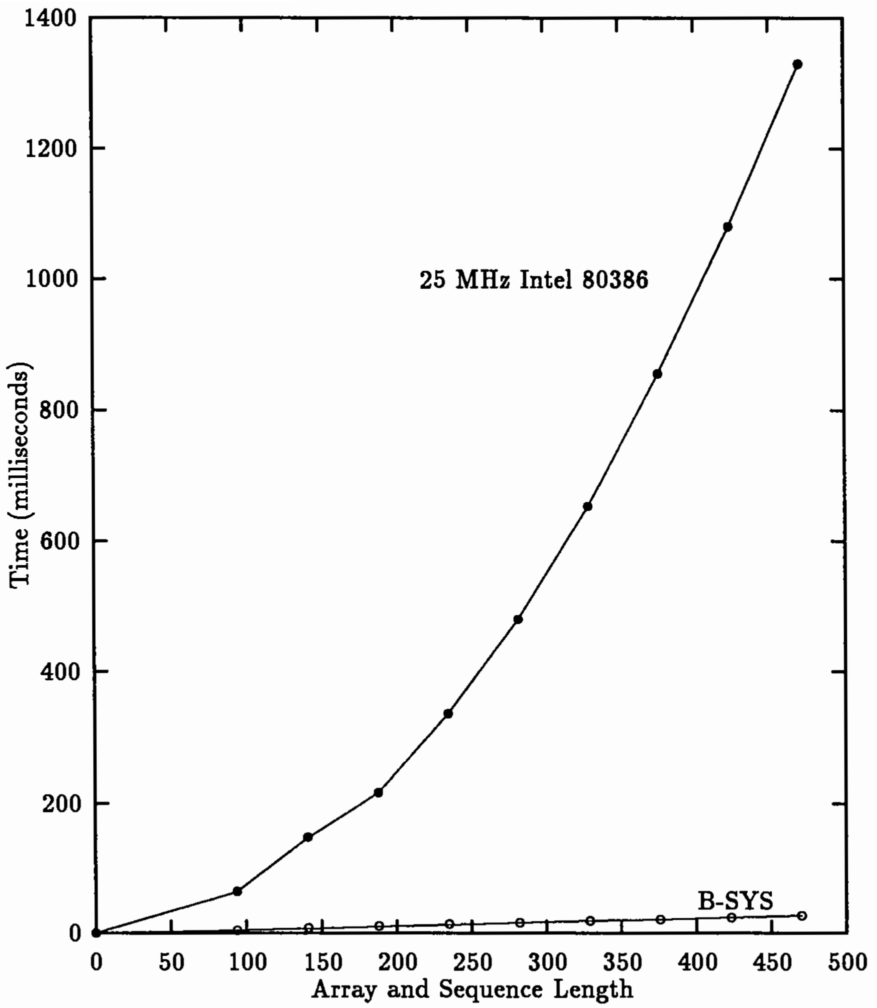


Figure 7.4: B-SYS one-against-one sequence comparison.

unchanged. The inclusion of the reset stream only requires one additional instruction, an instruction which concurrently shifts the reset stream through the array and modifies the modulo minimum cost selection according to the reset stream's value. Such efficiency would not be possible without the Systolic Shared Register implementation of systolic communication.

In Figure 7.6, two reset signals are sent through the array to preserve the correct systolic phasing, a complication which is avoided in NSL. Also note that each iteration of the bcompare() routine returns the cost of matching the stored sequence against the *previous* sequence sent into the array. Since only n cell program iterations are executed during each call to bcompare(), the current input sequence is left in the array, having not yet sent any data back to the host. As with the previous program, the raw modulo

```
#include "nsl.h"
#define DEL_COST 1
void
sequence (SStream& Char1, SStream& Char2, SStream& Cost, SStream& Reset)
{
  Flag f;

    // set f to bit 7 of the difference
  f = (Cost[-1] - Cost[+1]) << 1;

    // Reset == 0x00 or 0xff.  If Reset = 0xff, always choose
    // Cost[-1], otherwise use minimum of costs (mod 256).
  f = (Reset != 0) || f;

  Cost = select (match (Char1, Char2),
                 Cost,
                 select (f, Cost[-1], Cost[+1]) + DEL_COST);
}
```

Figure 7.5: NSL many-against-one sequence comparison.

output data is processed to extract the true $d_{n,n}$ value.

This optimized sequence comparison algorithm maintains 100% processor utilization, though only 99.6% of the computation involves the input strings: the remaining 0.4% of the computation involves the reset stream (these percentages are based on a 470-element array). This program can compare 470-character sequences on the B-SYS prototype system in 16.5 ms per sequence, corresponding to a speedup of 81.2 times faster than the 80386. Speedup ratios up to this point are displayed in Figure 7.7. A B-SYS* implementation with 470 processing elements could be expected to perform 470-character sequence comparison over 600 times faster than the 80386.

Performance Analysis

Comparing a 470-element systolic co-processor to a single microprocessor is not exactly fair. True evaluation of the Brown Systolic Array requires a performance comparison to a variety of machines. Fortunately, sequence comparison is a well-studied problem, in particular the four-character, many-against-one sequence comparison problem just described. Core and others have analyzed this problem on the Intel iPSC hypercube, Connection Machine (CM-1), and Encore Multimax.[30] Cheever and others have considered a signal processing approach to DNA correlation.[26] Lopresti has implemented the dynamic programming method on a variety of machines, including massively parallel and traditional supercomputers as well as systolic co-processors.[58,111]

The basic sequence comparison algorithm requires $O(n^2)$ resources, so that the performance of even the most powerful traditional supercomputer (with some small number of powerful central processing units) cannot surpass that of a simple systolic co-processor, even one with a slow bus interface such as the B-SYS prototype or P-NAC. Figure 7.8

```
int bcompare (register unsigned char *s,     /* Moving sequence        */
              int n,                         /* Array size             */
              register unsigned char *r)     /* Array of b-sys results */
{ int i;
  /* Send 2 reset signals into the array with 0 weights. */
  ZInstruction(1, xorABC, L2, R2, LF, Zsub, F7, F1);     /* Compare costs */
  ZInstruction(1, fnA, LF, LF, LF, Zmsb, F1, F1);        /* Check MSB     */
  Instruction(1, fnA, L6, L6, R6, 0xaf, F1, F1, 0xff);   /* Check reset   */
  ZInstruction(1, selectABonC, L2, R2, LF, Zconst, F1, F1); /* sel min    */
  ZInstruction(1, xorAC, LF, LF, LF, Zadda, F6, F1);     /* Add 1         */
  ZInstruction(1, fnA, L1, R0, R1, matchAB, F7, F2);     /* Compare char  */
  ZInstruction(1, selectABonC, L4, LF, R4, Zconst,F2,F2);
  ReadBsys (*r); r++;
  ZInstruction(1, xorABC, L4, R4, LF, Zsub, F7, F1);     /* Compare costs */
  ZInstruction(1, fnA, LF, LF, LF, Zmsb, F1, F1);        /* Check MSB     */
  Instruction(1, fnA, L6, L6, R6, 0xaf, F1, F1, 0xff);   /* Check reset   */
  ZInstruction(1, selectABonC, L4, R4, LF, Zconst, F1, F1); /* sel min    */
  ZInstruction(1, xorAC, LF, LF, LF, Zadda, F6, F1);     /* Add 1         */
  ZInstruction(1, fnA, L1, R0, R1, matchAB, F7, F2);     /* Compare char  */
  ZInstruction(1, selectABonC, L2, LF, R2, Zconst,F2,F2);
  ReadBsys (*r); r++;

  for (i = 0, _CX=0 ; i < n ; i+=2) {
    /* First phase */
    ZInstruction(1, xorABC, L2, R2, LF, Zsub, F7, F1);   /* Compare costs */
    ZInstruction(1, fnA, LF, LF, LF, Zmsb, F1, F1);      /* Check MSB     */
    ZInstruction(1, fnA, L6, L6, R6, 0xaf, F1, F1);      /* Check reset   */
    ZInstruction(1, selectABonC, L2, R2, LF, Zconst, F1, F1); /* sel min  */
    ZInstruction(1, xorAC, LF, LF, LF, Zadda, F6, F1);   /* Add 1         */
    Instruction(1, fnA, L1, R0, R1, matchAB, F7, F2, *s);/* Compare char  */
    Instruction(1, selectABonC, L4, LF, R4, Zconst,F2,F2,_CX);
    ReadBsys (*r); r++;
    _CX++; s++;
    /* Second phase */
    ZInstruction(1, xorABC, L4, R4, LF, Zsub, F7, F1);   /* Compare costs */
    ZInstruction(1, fnA, LF, LF, LF, Zmsb, F1, F1);      /* Check MSB     */
    ZInstruction(1, fnA, L6, L6, R6, 0xaf, F1, F1);      /* Check reset   */
    ZInstruction(1, selectABonC, L4, R4, LF, Zconst, F1, F1); /* sel min  */
    ZInstruction(1, xorAC, LF, LF, LF, Zadda, F6, F1);   /* Add 1         */
    Instruction(1, fnA, L1, R0, R1, matchAB, F7, F2, *s);/* Compare char  */
    Instruction(1, selectABonC, L2, LF, R2, Zconst,F2,F2,_CX);
    ReadBsys (*r); r++;
    _CX++; s++;
  }
  /* Reconstruct result of the previous sequence from modulo data */
  return (post_bsys (r, n));
}
```

Figure 7.6: B-SYS code for many-against-one sequence comparison.

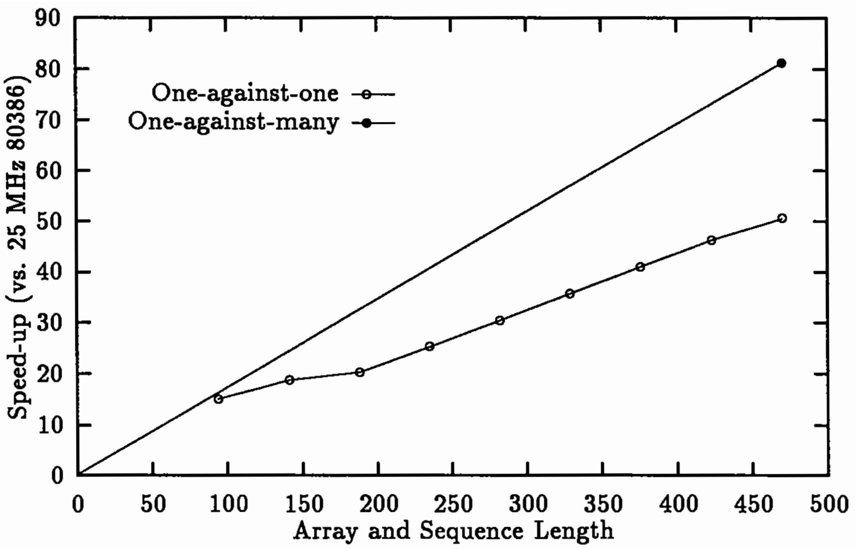


Figure 7.7: Two sequence comparison algorithms on B-SYS.

graphs the performance of several different machines on the many-against-one sequence comparison problem (execution times are per 100 sequences).[b] The timings for the Cray-2, Multiflow Trace, P-NAC, and Splash array are extrapolated from data on 100 comparisons of 100-character sequences.[58] For the first two machines, the extrapolation follows an $O(n^2)$ relationship, while for latter two the extrapolation follows an $O(n)$ relationship until the hardware bounds of the machine are reached, at which point execution time becomes $O(n^2)$ because the problem must be partitioned. The B-SYS timings are derived from the experimental results just described and the B-SYS* timings are estimated (both with a combined $O(n)$ and $O(n^2)$ relationship). The Connection Machine timings are more complicated, and are based on configuring the Connection Machine as a set of systolic pipelines of the appropriate length. The timings are from the experimental results presented in Appendix C.

The BioSCAN system has not been included in Figure 7.8 because a working system will not exist for at least another year. It can be expected to perform perhaps five times faster than B-SYS* (assuming that data can be rapidly passed to the BioSCAN chip), though only for sequence comparison without insertions or deletions or gap penalties but with programmable weights. Although potentially faster, it is not in the same league as B-SYS or Splash because it is not generally programmable — it is a more appropriate match to P-NAC.

As can be seen from Figure 7.8, B-SYS* is competitive: it is approximately twenty

[b] The figure is a logarithmic plot; the lines corresponding to the traditional computers have a slope of 2 (indicating an $O(n^2)$ execution time), while the systolic co-processors have a slope of 1 (indicating an $O(n)$ execution time) for sequences of equal or shorter length than the array. Vertical distances between curves indicate speedup factors between machines.

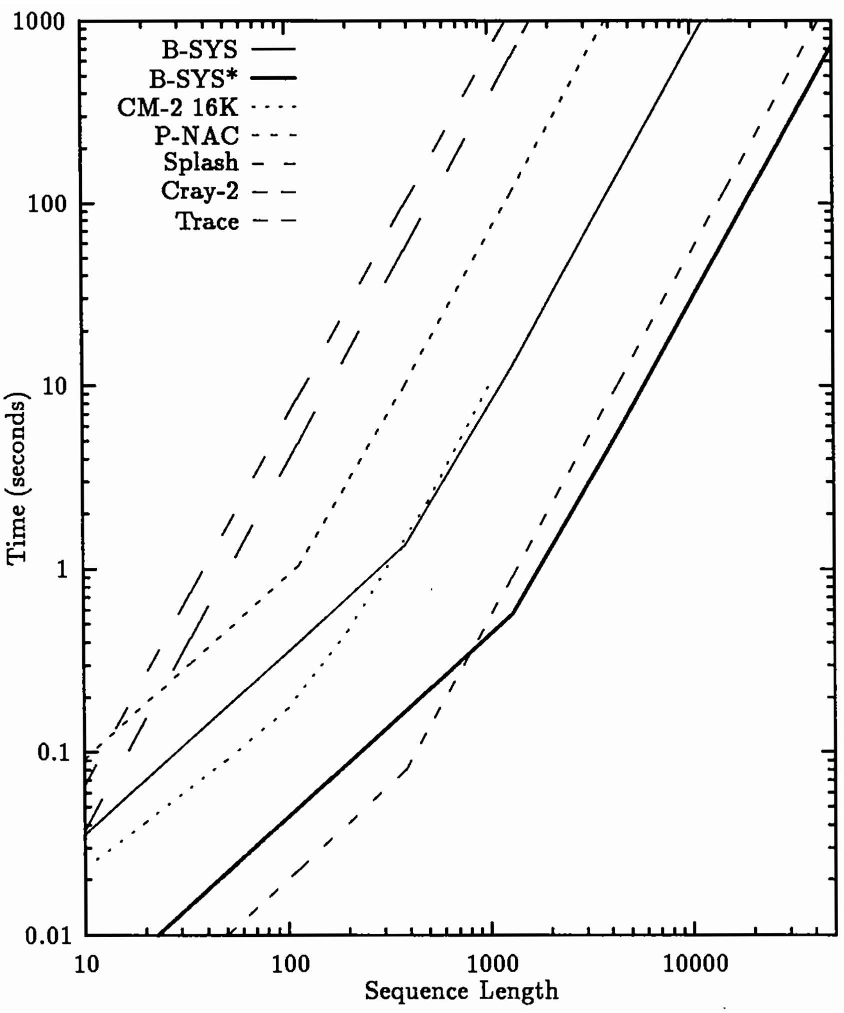


Figure 7.8: Sequence comparison on various machines.

times faster than the P-NAC system (indicating both a faster bus interface and a higher-performance VLSI implementation than P-NAC). The Splash programmable hardware system is twice as fast as B-SYS* on sequences shorter than 400 characters, but the higher density of processing elements on the single-board B-SYS* system lets it perform about 1.8 times faster than Splash on sequences with 1500 or more characters. With a redesigned 500-processor B-SYS chip, a 32-chip (16000-element) B-SYS system would perform 2500 times faster than P-NAC and 20 times faster than Splash when $n > 16000$.

Recall also that B-SYS features *general* programmability: P-NAC's algorithm cannot be varied at all, while slight variations to the Splash algorithm will result in a much lower number of processing elements per Splash board. Programming the machines for 7-bit character comparison will have no effect on the number of available B-SYS functional units, but the number of systolic cells on the Splash system will drop by one third, moving Splash's bend between $O(n)$ and $O(n^2)$ performance to $n = 250$, increasing the performace factor difference to 2.7 on sequences with 1500 or more characters. More complicated systolic algorithms (which cannot be handled at all by P-NAC or BioSCAN) will produce an even starker contrast between Splash and B-SYS.

The ease of programmability is also a concern. For this problem, P-NAC requires no lines of code, B-SYS requires tens of lines of code, and the Connection Machine and other supercomputers, without automatic support for systolic streams, require slightly longer programs. The Splash system, on the other hand, requires nearly 3000 lines of code to perform 4-bit sequence comparison.[112] With the development of NSL, the programming and testing of new B-SYS applications requires little additional time beyond that required for selection of an appropriate systolic algorithm.

Thus, in addition to providing a higher concentration of systolic processing power than Splash, B-SYS is much more readily programmable, allowing rapid testing of new algorithms and metrics.

Linear systolic arrays are eminently scalable; the formation of large linear arrays has no inherent limitation, unlike I/O-bound meshes or complex hypercube networks. A ten-thousand-element B-SYS* machine would require approximately seven boards, certainly a reasonable size when one considers the complexity of the Connection Machine. A Splash system, unable to pack systolic cells as tightly, would require almost thirty boards to compare 10 000-character sequences drawn from a 4-character alphabet. The CM-2, however, cannot be easily scaled beyond its 64 K processing elements because its message routers only support 16-dimensional hypercubes. (It is possible to compare long strings on short arrays, but not as efficiently or quickly.[108])

Thus, execution time alone is not a just figure of merit, and the use of resources, both time and space, should be considered. To solve a specific problem as quickly as possible, the length of a linear systolic array should match the size of the data. Given this match, a specific amount of resources will be required to solve the problem. Figure 7.9 graphs the product of system size and execution time for the massively parallel machines. As a rough estimate of system size, the approximate number of chips in each system has been used. To compare sequences of a specific length, B-SYS* requires 4 times fewer resources than Splash, a factor which will quickly rise as more complicated sequence comparison algorithms are programmed on the two machines. When complete, the BioSCAN system could be expected to have a resource measure many times better than that of B-SYS* due to its high concentration of single-purpose processing elements. A more aggressive VLSI implementation of the B-SYS chip would improve the B-SYS* values tenfold. The Connection Machine, with its large amount of hardware (over 4 K chips), is no longer an attractive solution to the sequence comparison problem: its high complexity overcomes the advantages of its massive parallelism.

It is hoped that this discussion of sequence comparison has left the reader convinced that although B-SYS can be slower than single-purpose systolic arrays on simple problems, the power of programmability more than makes up for the small difference in performance. Having examined sequence comparison performance in great detail, the remaining applications will only discuss implementations on the B-SYS system, for which

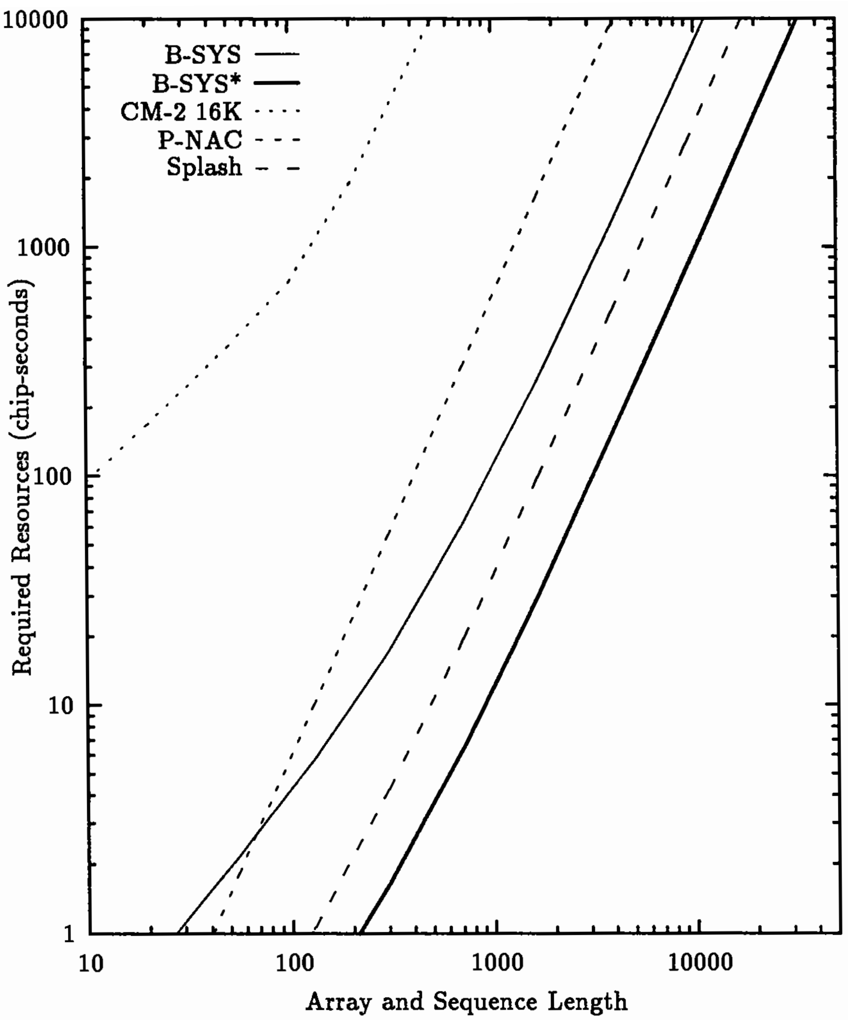


Figure 7.9: Sequence comparison on various machines (resources).

cell-program length is an accurate gauge of execution time (each instruction averages 4.35 ms on B-SYS and an estimated 0.54 ms on B-SYS*).

7.1.3 Sequence Comparison with Gap Penalties

Gap penalties are used by computational biologists who feel that the deletion or insertion of *any* number of nucleotides has an extra cost for the introduction of a gap in the sequence (computational biologists are by no means unanimous about what metrics should be used when comparing sequences). For example, the strings AAUUUC and AUUAUC both have a standard edit distance of 3 from AAC, however the latter breaks the sequence twice in its insertion of three uracil nucleotides, and thus could be considered the more distant from AAC. Applying a gap penalty of 2 will reflect this difference: the costs would become 5 and 7 with the introduction of the gap penalty, indicating that AAUUUC is more similar to AAC than AUUAUC.

In general, molecular biologists assign a ***higher*** cost to gap creation than to point mutation, but an incrementally lower cost to the continuation of gaps. In evolutionary terms, it is difficult (costly) to break a nucleic acid, but once a break has occurred the cost of continuing that break is not high. For sequence comparison with gap penalties, the cost of deleting n contiguous nucleotides will be

$$\text{gap}(a_1) + \sum_{i=1}^{n} \text{dist}(\phi, a_i),$$

where the penalty for starting a gap by inserting or deleting the character c is $\text{gap}(c)$. For nucleic acid research, typical values are a mutation cost

$$\text{dist}(c_1, c_2) = 1 \;\; \forall c_1 \neq c_2,$$

a gap introduction cost $\text{gap}(c)$ between 3 and 5, and an incremental deletion or insertion cost $\text{dist}(c, \phi) = 0.1$ (when comparing nucleic acids, gap, mutation, and insertion or deletion costs are character invariant, leading to the affine deletion cost of $n \cdot \text{dist}(\phi, c) + \text{gap}(c)$).[159]

Sequence comparison with gap penalties is somewhat complicated and is thus a good indicator of the power of the B-SYS programmable systolic array. The recurrence equations for this type of sequence comparison are on three variables, f, g, and h, and cost values are obtained by minimizing these three variables.[85]

$$\begin{aligned} f_{0,0} &= \text{gap}(a_1) \\ g_{0,0} &= \text{gap}(b_1) \\ h_{0,0} &= 0 \end{aligned} \tag{7.1}$$

$$\begin{aligned} f_{i,0} &= f_{i-1,0} + \text{dist}(a_i, \phi) \\ g_{i,0} &= g_{i-1,0} + \text{dist}(a_i, \phi) \\ h_{i,0} &= f_{i,0} \end{aligned} \tag{7.2}$$

$$\begin{aligned} f_{0,j} &= f_{0,j-1} + \text{dist}(\phi, b_j) \\ g_{0,j} &= g_{0,j-1} + \text{dist}(\phi, b_j) \\ h_{0,j} &= f_{0,j} \end{aligned} \tag{7.3}$$

$$f_{i,j} = \text{dist}(a_i, \phi) + \min \begin{cases} f_{i-1,j} \\ g_{i-1,j} + \text{gap}(a_i) \\ h_{i-1,j} + \text{gap}(a_i) \end{cases} \tag{7.4}$$

$$g_{i,j} = \text{dist}(\phi, b_j) + \min \begin{cases} f_{i,j-1} + \text{gap}(b_j) \\ g_{i,j-1} \\ h_{i,j-1} + \text{gap}(b_j) \end{cases} \tag{7.5}$$

$$h_{i,j} = \text{dist}(a_i, b_j) + \min \begin{cases} f_{i-1,j-1} \\ g_{i-1,j-1} \\ h_{i-1,j-1} \end{cases} \tag{7.6}$$

f	ϕ	A	A	C
ϕ	2	3	4	5
A	3	4	5	6
U	4	3	6	7
U	5	4	5	8
A	6	5	6	7
U	7	6	7	8
C	8	7	8	9

g	ϕ	A	A	C
ϕ	2	3	4	5
A	3	4	3	4
U	4	5	6	5
U	5	6	7	8
A	6	7	8	7
U	7	8	9	8
C	8	9	10	11

h	ϕ	A	A	C
ϕ	0	3	4	5
A	3	0	3	6
U	4	3	2	5
U	5	6	5	4
A	6	5	4	7
U	7	8	7	6
C	8	9	8	7

Figure 7.10: Dynamic programming matrices for gap sequence comparison.

$$d_{i,j} \quad = \quad \min \begin{cases} f_{i,j} \\ g_{i,j} \\ h_{i,j} \end{cases} \tag{7.7}$$

In the computation of a specific $d_{i,j}$ value, the h values determine the cost of mutating the character a_i to b_j, while the f and g values determine the cost of introducing or continuing a gap in the A or B sequences, respectively. The comparison of AAC with AUUAUC using $\text{gap}(c) = 2$, $\text{dist}(c_1, c_2) = 2$, and $\text{dist}(c, \phi) = 1$ yields the dynamic programming matrices of Figure 7.10.

An NSL program for gap sequence comparison using a gap penalty of 3, insertion or deletion cost of 0.1, and point mutation cost of 1 is presented in Figures 7.11 and 7.12. Because decimals are messy, the costs have been scaled by a factor of 10. In spite of this factor, differences between neighboring costs remain less than 128, so comparison modulo 256 is valid. Again, the modulo sequence comparison data must be processed (in linear time) to determine the true $d_{n,n}$ value. The NSL program produces a 24-line B-SYS program, and gap sequence comparison will take four times longer than basic one-against-one sequence comparison and 3.9 times longer than basic many-against-one sequence comparison on addition of a reset stream.

7.1.4 Alternate Transcription[c]

The *alternate transcription problem* is a generalization of sequence comparison in which there are several equally acceptable target sequences. One of the earliest studied alternate transcription problems was the matching of a sequence to a context-free grammar, with the motivation being the construction of error-correcting parsers.[4,114] Wagner surveys the complexity of various formal language error correction problems and proves the context-sensitive case to be NP-hard.[155] The redundancy in the meaning and pronunciation of words makes speech recognition another possible application.

Wagner has proposed a dynamic programming algorithm for comparing a regular language and a string.[156] Kruskal and Sankoff describe a dynamic programming algorithm for comparing the languages of two finite state automata without feedback, which

[c] This section extends part of the work which lead to the author's master's degree.[68]

```
#include "nsl.h"
#define DEL_COST 1
Register
modmin (Register& x, Register& y)
{
  Flag f;
  f = (x - y) << 1;
  return (select (f, x, y));
}
void
gapsequence (SStream& Char1, SStream& Char2,
             SStream& Costf, SStream& Costg, SStream& Costh,
             SStream& Cgap,  SStream& Cmutate)
{
  Costh = modmin (Costf, modmin (Costg, Costh));
  force_mask (nomatch (Char1, Char2));
     Costh.out() = Costh.out() + Cmutate;
  force_mask(1);

  Costf = modmin (Costf[+1],
                  modmin (Costg[+1], Costh[+1]) + Cgap) + DEL_COST;
  Costg = modmin (Costg[-1],
                  modmin (Costf[-1], Costh[-1]) + Cgap) + DEL_COST;
}
```

Figure 7.11: NSL cell program for gap sequence comparison.

they refer to as *networks*.[85] A network for four code words is displayed at the top of Figure 7.13. When two or more paths branch out from a node in the network, a choice must be computed. Two or more paths entering a node indicate the minimization operation of picking the best choice from a previous branching. The network version of the sequence comparison recurrence (Equations 2.8–2.11 on page 12) for a string A and network B is:

$$d_{0,0} = 0 \tag{7.8}$$

$$d_{i,0} = d_{i-1,0} + \text{dist}(a_i, \phi) \tag{7.9}$$

$$d_{0,j} = d_{0,j-1} + \text{dist}(\phi, b_j) \tag{7.10}$$

$$d_{i,j} = \min \begin{cases} \min_{j^*}\{d_{i-1,j^*} & + \text{dist}(a_{i-1}, b_{j^*})\} \\ d_{i-1,j} & + \text{dist}(\phi, b_j) \\ \min_{j^*}\{d_{i,j^*} & + \text{dist}(a_i, \phi)\}. \end{cases} \tag{7.11}$$

The notation j^* (from Kruskal and Sankoff) refers to all the predecessors of the current element $d_{i,j}$ in the second (columnar) coordinate, or all the nodes with edges entering node j of the network. Note that there may be several different $d_{i,j}$ values in the network, corresponding to all nodes in B that are a distance j from the source (root) node. When comparing two networks, the $(i-1)$ terms in Equation 7.11 are replaced by i^*, and the first two cases of Equation 7.11 are minimized over i^*. The initial conditions for network

```
#include "nsl.h"
int
fgweight (const int n)
{
  return ((n + 30) % 256);    // = n*dist(c,0)+gap(c)
}
int
hweight  (const int n)
{
  return (n == 0 ? 0 : fgweight(n));
}
main(void)
{
  const int n = 5;
  int f[n], g[n], h[n];
  Config::setup (n, 16, 8, 2, 1, LINEAR, 8);

  // Two constants of 10 and 30 for mutations and
  // gaps (insert/delete assumed to be 1)
  FixSStream Cmutate (INT, 1, 10);
  FixSStream Cgap    (INT, 1, 30);

  SStream    Seq1 (EAST, NSL_SPEED0, CHAR, NSL_BYTE1);
  Seq1.source     ("seq1", n);

  SStream    Seq2 (EAST, NSL_SPEED1, CHAR, NSL_BYTE1);
  Seq2.source     ("seq2", n);

  SStream fWeight (EAST, NSL_SPEED2, INT, NSL_BYTE1);
  fWeight.source  (fgweight, n);
  fWeight.sink    (f, n, IDENT, n);

  SStream gWeight (EAST, NSL_SPEED2, INT, NSL_BYTE1);
  gWeight.source  (fgweight, n);
  gWeight.sink    (g, n, IDENT, n);

  SStream hWeight (EAST, NSL_SPEED2, INT, NSL_BYTE1);
  hWeight.source  (hweight, n);
  hWeight.sink    (h, n, IDENT, n);

  store_code   ("GapSequence", gapsequence, &Seq1, &Seq2,
                &fWeight, &gWeight, &hWeight, &Cgap, &Cmutate);

  execute_code ("GapSequence", &Seq1, &Seq2,
                &fWeight, &gWeight, &hWeight, &Cgap, &Cmutate);

  post_process (f, g, h, n);
}
```

Figure 7.12: NSL main program for gap sequence comparison.

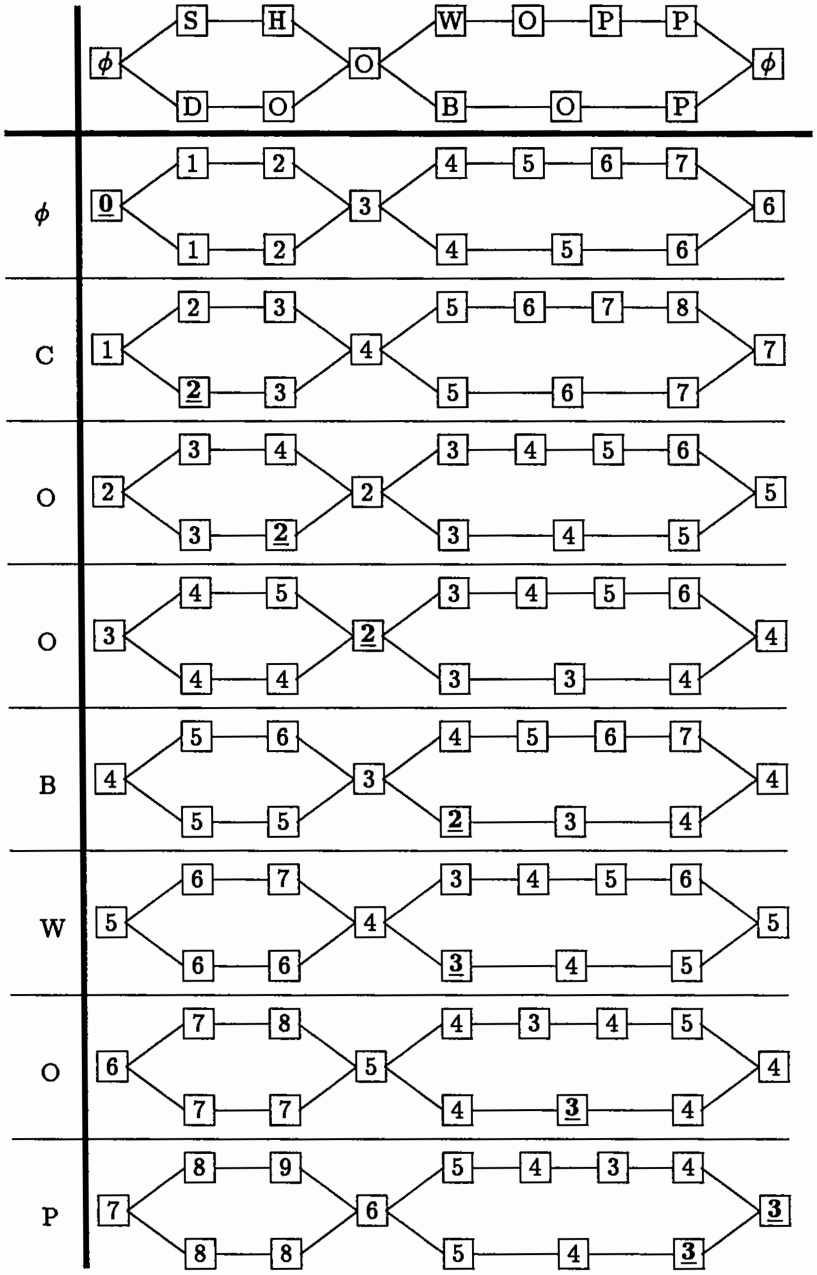

ϕ S H D O O W O P P B O P ϕ
ϕ 0 1 2 1 2 3 4 5 6 7 4 5 6 6
C 1 2 3 2 3 4 5 6 7 8 5 6 7 7
O 2 3 4 3 2 2 3 4 5 6 3 4 5 5
O 3 4 5 4 4 2 3 4 5 6 3 3 4 4
B 4 5 6 5 5 3 4 5 6 7 2 3 4 4
W 5 6 7 6 6 4 3 4 5 6 3 4 5 5
O 6 7 8 7 7 5 4 3 4 5 4 3 4 4
P 7 8 9 8 8 6 5 4 3 4 5 4 3 3

Figure 7.13: Network comparison.

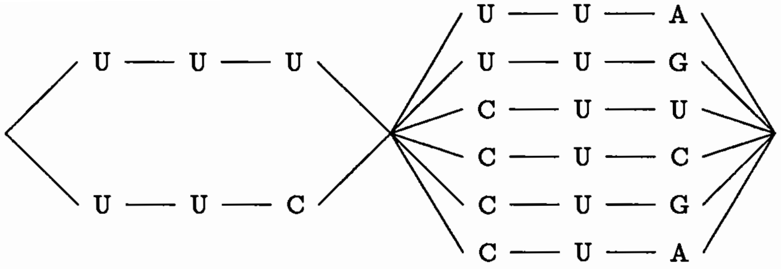


Figure 7.14: Comparison network for phenylalanine—leucine.

comparison are similar to those of the basic algorithm, and the zeroth row or column is a super-source for the network.

Network sequence comparison is rather involved, and an example is in order. Consider a code used by a clan of secret agents that consists of the words SHOBOP, SHOWOPP, DOOBOP, and DOOWOPP. One day, a message with the word COOBWOP is received. Since all messages are of vital importance, it must be decoded. Not finding the word in the code book, the agent must determine which word the writer most likely meant. This is solved with string comparison and network comparison is more efficient than comparing COOBWOP to each of the four words in the code. The network and the associated dynamic programming calculations are shown in Figure 7.13. Looking at the places where minimization occurs (the third letter and the final ϕ), it is evident which branches were chosen. The closest match to the incorrect code word is DOOBOP, and the edit distance between DOOBOP and COOBWOP is three, with a mutation and a deletion. The minimizing route is indicated with boldface entries.

Protein encoding regions

A code of more pressing concern than that of the secret agents is the genetic code of Table 1.1 on page 3, in which each amino acid is encoded by up to six different codons (sets of three nucleotides). Given a desired amino acid sequence (corresponding to a specific protein), the problem is to determine its similarity to a nucleic acid. Peltola has used an approximate dynamic programming solution[131] (an analysis of this approximation has been previously presented by this author[68]), and neural network solutions have also been proposed.[44] Subsequence variations on this problem can locate protein encoding regions within a nucleic acid. This section describes an exact dynamic programming algorithm and its implementation on the B-SYS programmable systolic array.

First, it should be noted that the translations of individual amino acids are independent. Consider the 2-element amino acid sequence of phenylalanine followed by leucine. The translation of phenylalanine to a 3-nucleotide codon in no way affects the possible translations of leucine to a codon. Thus, the basic network has a width of at most six (the highest level of redundancy in the genetic code), as displayed in Figure 7.14. A careful analysis of the genetic code, however, can simplify the basic network considerably. In all

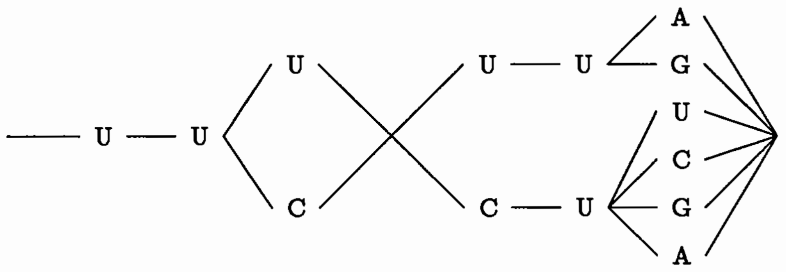


Figure 7.15: Uniform comparison network for phenylalanine—leucine.

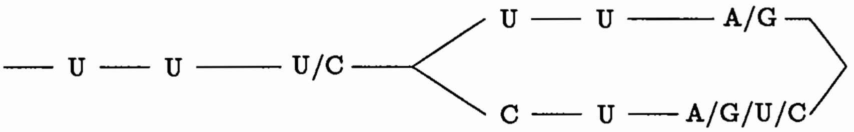


Figure 7.16: Wildcard comparison network for phenylalanine—leucine.

cases, there is at most one pair of 2-nucleotide prefixes for every codon encoding of an amino acid. Thus, the initial segments of an amino acid network can be compressed, as seen in Figure 7.15.

Further simplification of the phenylalanine and leucine network is achieved with the wildcard encoding commonly used for nucleotides. Four bits are used to represent each nucleotide, encoding them as A = 1000, C = 0100, G = 0010, and U = 0001. Two characters match if they have any corresponding asserted bit positions, as was checked in previous examples with the B-SYS match instruction. Thus, the wildcard character 0101 will match both C and U (called the pyrimidines), 1010 will match both A and G (the purines), and 1111 will match all four nucleotides. Using wildcards, the network is further compressed to have a maximum width of 2, as illustrated in Figure 7.16. A close examination of Table 1.1 on page 3 will reveal that all amino acids can be represented with width-2 wildcard networks.

Implementation of this algorithm on a programmable systolic array is simple. The two sequence comparison tracks of the network are evaluated sequentially. After the completion of each pair of node updates, every third processing element (corresponding to the last nucleotide of a codon) selects the minimum of its two tracks to pass to the next processing element. An NSL cell program for this algorithm can be seen in Figure 7.17. Note that it uses the basic sequence comparison cell program seen earlier, and could just as easily use a reset stream as well. Local cost differences are less than 128, so single-byte modulo comparison is used. The main program (not shown) will generate and store the wildcard network in the fixed `Char1a` and `Char1b` systolic streams. The nucleotide sequence is passed through the array in the mobile `Char2` stream. The fixed `Third` stream identifies every third processing element for the conditional minimization. As usual, the

```
#include "nsl.h"
#define DEL_COST 1
void
sequence (SStream& Char1, SStream& Char2, SStream& Cost)
{
  Flag f;
  f = (Cost[-1] - Cost[+1]) << 1;

  Cost = select (match (Char1, Char2),
                 Cost,
                 select (f, Cost[-1], Cost[+1]) + DEL_COST);
}
void
alt_trans (SStream& Char1a, SStream& Char2, SStream& Costa,
           SStream& Char1b,                 SStream& Costb,
           SStream& Third);
{
  sequence (Char1a, Char2, Costa);
  sequence (Char1b, Char2, Costb);

  Flag f;
  // Third is a FixSStream, = 0xff for every third PE.
  force_mask (Third != 0);
     f = (Costa.out() - Costb.out()) << 1   // Min mod 256
     Costa.out() = Costb.out() = select (f, Costa.out(), Costb.out());
  force_mask (1);
}
```

Figure 7.17: NSL amino acid encoding comparison.

movement of all systolic streams is not processed until the entire cell program (in this case, alt_trans()) has been evaluated. A B-SYS cell program for this problem has 16 instructions, requiring only 2.7 times more time than the simple sequence comparison program.

This example highlights the major steps of systolic algorithm development. First, the network algorithm was developed and refined for this particular problem, leading to the simple computational requirements of the sample cell program. Optimization of this program for the B-SYS hardware, making full use of the Systolic Shared Register architecture, lead to a cell program with no overhead: each instruction performs a required operation and there are no instructions devoted solely to data movement.

7.1.5 Protein Matching

As for nucleic acid analysis, sequence comparison is vital for protein analysis. Proteins tend to be shorter than nucleic acids, with lengths in the thousands or tens of thousands of amino acids. This section considers a B-SYS program for the protein scanning algorithm implemented by BioSCAN. It is properly called an alignment or diagonal scan algorithm since it does not allow insertions or deletions; when comparing two sequences A and B of lengths m and n ($m > n$), the cost of matching B to a subsequence of n contiguous characters in A is determined. The diagonal scan information may be statistically analyzed to determine an appropriate matching between the sequences.[77] Although not as rigorous as full-fledged sequence comparison, this alignment algorithm is a valid filter for databases searches.

The BioSCAN system can align sequences over any 28-character alphabet, but was particularly designed for protein sequence analysis. BioSCAN implements the recurrence:

$$
\begin{aligned}
d_{0,0} &= 0 && (7.12)\\
d_{i,0} &= 0 && (7.13)\\
d_{0,j} &= d_{0,j-1} + \mathrm{dist}(\phi, b_j) && (7.14)\\
d_{i,j} &= d_{i-1,j-1} + \mathrm{dist}(a_i, b_i). && (7.15)
\end{aligned}
$$

This is a subsequence search: the system is used to find regions in a long string (A) which are similar to the entire shorter string (B). Thus, there is no penalty for starting at any position in A, as is reflected by Equation 7.13. However, to find a complete copy of B, the $d_{0,j}$ initialization values from the simple sequence comparison recurrence are maintained. The region of highest similarity to the B string is located by minimizing all $d_{i,n}$ values, since the subsequence alignment is allowed to end at any point within the A string. In general, A is a collection of sequences from a database and B is the sequence of specific interest.

In contrast to nucleic acid sequence comparison, protein sequence comparison does not use a single mutation cost. Some amino acids mutate less readily than others and are assigned correspondingly higher mutation costs. Thus, a four-hundred-element (20×20) table of mutation costs among the 20 amino acids is required. Biologists most often use the PAM-250 cost table proposed by Dayhoff and others when comparing proteins.[32] These costs are scaled logarithms of the probabilities of point mutations from one amino acid to another. That is, for a pair of amino acids, if it were experimentally determined that the first mutates to the second with some low probability p, a value proportional to $-\log(p)$ represents the high cost of matching the two amino acids (since a match would require an unlikely mutation to take place). The summing of these logarithmic weights corresponds to the multiplication of probabilities as the diagonal scan is computed.

For the systolic algorithm that assigns characters from one sequence to individual processing elements, each processing element need only store the twenty costs corresponding to a single column of the table. The corresponding character does not need to be stored; the 20 costs are sufficient. Unfortunately, each B-SYS register bank has only 16 registers. The solution to this problem is to couple pairs of functional units to represent each character in the B sequence, and as it turns out this will produce a very efficient algorithm: each pair of functional units will concurrently locate the correct cost, fully utilizing all functional units and memory banks. If many processing elements are available, the characters can be divided among three or more cells for an even faster algorithm.

An NSL program for protein alignment is displayed in Figures 7.18 and 7.19. The

```
#include "nsl.h"
void
psequence (SStream& Char1, SStream& Cost,
           SStream& Table, SStream& Base)
{
  Register TrialCost (INT, 1, EAST);
  Register TrialChar (CHAR, 1);

  TrialCost = 0;
  TrialChar = Base;

  // Search for a match
  int i = 0;
  do {
    TrialCost = select (TrialChar == Char1,
                        Table.in().byte(i), TrialCost);
    TrialChar++;
  }  while (++i < 10);

  // Combine costs to give both PEs same cost.
  force_mask (Base == 0); {
    TrialCost = TrialCost + TrialCost.opposite_reg();
    TrialCost.opposite_reg() = TrialCost;
  } force_mask (1);

  // Shift streams twice
  Cost += TrialCost;
  Cost = Cost;

  Char1 = Char1;
  Char1 = Char1;
}
```

Figure 7.18: NSL protein alignment cell program.

program exhibits several new NSL functions. Each pair of adjacent processing elements is assigned one character, and each member of the pair is assigned half of the appropriate cost table. Which set of 10 costs a functional unit is responsible for is indicated by the `Base` stream. The characters are numbered consecutively from 0 to 20, and the `Base` stream places values of 0 and 10 in adjacent function units. The loop of Figure 7.18 increments the base number and checks for a character match. If they do match, the appropriate cost from the table is stored in the `TrialCost` register — the `byte()` function accesses a specific byte from a multi-byte NSL register. Since the character will only match in one of the two neighboring functional units, only one `TrialCost` is non-zero for each pair of functional units at the end of the loop. The two `TrialCost` values are added together by the leader of the pair (in this program, the eastern member of the pair) and the result is then copied to the `TrialCost` register of both members of the pair: since

```
#include "nsl.h"
const int n = 200;
int weights[20][20];
int fixed_string[n];
int base (const int n)
{
  return ( (n % 2 == 0) ? 10 : 0);
}
int tablecolumn (const int n)
{
  return (weights[fixed_string[n/20]][n%20]);
}
main(void)
{
  int d[2*n];
  Config::setup (2*n, 16, 8, 2, 1, LINEAR, 8);

  FixSStream Base (INT,1);
  Base.source (base, n);

  read_file_into_array ("seq1", fixed_string, n);
  read_file_into_array2 ("cost_table", weights, 20, 20);

  SStream    Seq1 (EAST, NSL_SPEED1, CHAR, NSL_BYTE1);
  Seq1.force_weave (NOWEAVE);
  Seq1.source     ("seq2", n);

  // Default source is 0
  SStream Cost (EAST, NSL_SPEED1, INT, NSL_BYTE1);
  Cost.force_weave (NOWEAVE);
  Cost.sink    (d, 2*n, IDENT, 2*n);

  FixSStream Table (INT, 10);
  Table.source (tablecolumn, n);

  store_code   ("ProteinSequence", psequence, &Seq1,
                &Cost, &Table, &Base);
  execute_code ("ProteinSequence", &Seq1,
                &Cost, &Table, &Base);
  post_process (d, n);
}
```

Figure 7.19: NSL protein alignment main program.

TrialCost is stored in register E_2 (for example) of $\mathcal{F}_1$, $\mathcal{F}_0$'s TrialCost is stored in $\mathcal{F}_1$'s W_2 register, referred to by the opposite_reg() member function of the *Register* class. This value is then added to the cost data stream. Both mobile data streams are then shifted twice to pass data to the next pair of processing elements (this would not be required with NSL implementation enhancement number 3 on page 110).

In the main program (Figure 7.19), initialization functions for the Base stream and the cost table are declared. Also, weaving is disabled for the mobile cost and sequence streams. Normally, since the streams are referred to after they are set during the shifting process, NSL would weave both streams. Since this algorithm requires the streams to be shifted twice, weaving must be disabled for proper code generation. Finally, as usual, the output data is processed to recover the exact diagonal score from the modulo data.

The cell program for protein alignment has 41 lines, and requires 204 ms per iteration on B-SYS or 25.4 ms per iteration on B-SYS*.

7.2 Matrix Multiplication

Matrix multiplication and the generalized algebraic path problem are the basis of many problems: transitive closure, all-pairs shortest path, and the dynamic programming recurrence for optimal parenthesization and RNA folding.[3, 135, 153] Multiplication of $n \times n$ matrices is not a problem normally associated with linear systolic arrays; algorithms for square and hexagonal meshes are common, and provide an $O(n)$ execution time on $O(n^2)$ processing elements. Nevertheless, linear systolic arrays are an attractive solution to matrix multiplication because of their fixed I/O requirements. The mesh algorithms, as can be inferred from their asymptotic processing time, require $O(n)$ inputs per time step, a potential problem for sequential storage devices. On the other hand, linear systolic arrays have a fixed number of inputs or outputs regardless of the length of the array. Linear systolic solutions to the matrix multiplication problem require either $O(n)$ processing elements, each with $O(n)$ words of memory, or $O(n^2)$ processing elements, each with $O(1)$ words of memory; these bounds are a result of the complex data dependencies which require data streams with $O(n)$ delay elements.

This section considers the general $O(n^2)$ matrix multiplication algorithm proposed by Ramakrishnan and Varman which is independent of the size of the input array.[133] For any given value of n, it is possible to derive simpler systolic algorithms, ones without such complex control, following the methods of Lee and Kedem.[102] The algorithm presented herein, however, does not require remapping to the systolic array for each different input size, though additional processing elements are required. The algorithm will be given with minimal explanation; the reader is referred to the previously cited article for further information.

Figure 7.20 illustrates the basic systolic cell for this algorithm. In addition to the A, B, and C matrix streams, there are two control streams and a second stream associated with the A matrix. As illustrated, the B and the *Cntrl1* streams move at a speed of 2, while the remaining streams move at one processing cell per step. The *AValid* stream is either equal to the A stream or to zero. When the control signals activate an A value, it is

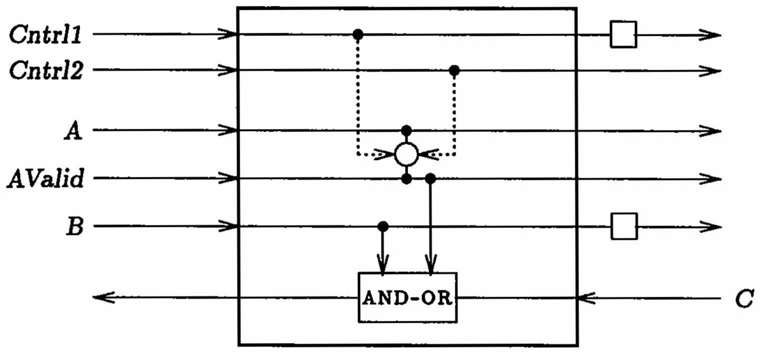


Figure 7.20: Systolic transitive closure cell.

```
#include "nsl.h"
void
inner (SStream& Avalid, SStream& B, SStream& C,
       SStream& A, SStream& Cntrl1, SStream& Cntrl2)
{
  force_mask (Cntrl1 == Cntrl2);
     Avalid.in() &= Cntrl1;       // Deactivate A input if required.
  use_mask (FALSE);               // Stop using mask
     C = C | (Avalid&B);
  use_mask (TRUE);
     Avalid.in() = A & Cntrl1;    // Possibly copy to active stream.
}
```

Figure 7.21: Transitive closure cell program.

transferred to the *AValid* track, and is later removed when the control signals deactivate the value. Activation occurs whenever a *Cntrl1* and *Cntrl2* activate signal meet in the same processor (FF is used in the NSL implementation), and similar conditions deactivate the *AValid* stream (when 00 is present in both control streams). The basic algorithm is:

```
if (Cntrl1 == 0xff && Cntrl2 == 0xff)
  Copy A to AValid
else if (Cntrl1 == 0x00 && Cntrl2 == 0x00)
  Clear AValid
else
  Output C = C | (A&B)
```

This is illustrated by the cell program of Figure 7.21, with a few slight modifications for efficiency. NSL converts this cell program into a simple 10-line B-SYS program with two phases. The main program (Figure 7.22) is straightforward, excepting the routines for correctly timing the inputs and outputs.

In spite of the simple cell program, the macroscopic data movement of this algorithm

```
main(void)
{
  Config::setup (array_size, 16, 8, 2, 1, LINEAR, 8);
  SStream MatA   (EAST, NSL_SPEED1, INT, NSL_BYTE1);
  SStream MatB   (EAST, NSL_SPEED2, INT, NSL_BYTE1);
  SStream MatC   (WEST, NSL_SPEED1, INT, NSL_BYTE1);
  SStream Avalid (EAST, NSL_SPEED1, INT, NSL_BYTE1);
  SStream Cntrl1 (EAST, NSL_SPEED2, INT, NSL_BYTE1);
  SStream Cntrl2 (WEST, NSL_SPEED1, INT, NSL_BYTE1);

  // Give inputs and take outputs forever.
  MatA.source    (Ainput,   maxtime);
  MatB.source    (Binput,   maxtime);
  MatC.sink      (Coutput,  maxtime);
  Cntrl1.source  (control1, maxtime);
  Cntrl2.source  (control2, maxtime);

  store_code ("Transitive", inner, &Avalid, &MatB, &MatC,
      &MatA, &Cntrl1, &Cntrl2);
  execute_code ("Transitive", &Avalid, &MatB, &MatC,
      &MatA, &Cntrl1, &Cntrl2);
}
```

Figure 7.22: Transitive closure main program.

is quite complex. A linear array of size

$$n^2 + (n-1) \cdot \left(\left\lfloor \frac{n}{2} \right\rfloor + 1 \right) + 2$$

is required for the algorithm; as mentioned, there are more efficient mappings of matrix multiplication or transitive closure to linear systolic arrays for specific values of n. Lee and Kedem use as an example $n = 4$ matrix multiplication which, for their mapping, requires ten processing cells. The Ramakrishnan and Varman algorithm requires 27 cells when $n = 4$, but has the advantage of data movement which is independent of problem size. Ideally, since the major area requirement of this algorithm is for delay elements, a mapping should be chosen which uses as much memory as possible in each processing element to keep the leading coefficient of the $O(n^2)$ processor requirement as low as possible.

The spacing of inputs and outputs for the co-processor depends on the size of the matrices being multiplied. An inefficient implementation of these equations is displayed in Figure 7.23. Each cell iteration index is broken down into the proper i and j values for the matrix indices. The important thing to note is the ease with which the complex algorithm description has been converted into a systolic program. Although further analysis of the algorithm could lower the overhead associated with calculating each input, an NSL program for this application can be generated by copying the algorithm description with minimal change. Thus, using NSL published algorithms can be implemented with

```
const int n=16;
const int r = n/2;
const int array_size = (n-1) * r + n*n + 2;
const int maxtime = array_size * 2;
int A[n][n];
int B[n][n];
int C[n][n];

#define inside(x,y,z) ( ((x) >= (y)) && ((x) < (z)))
int control1 (const int t)
{
  if ((t-2) % (3*r+3) == 0)
    return (inside((t-2)/(3*r+3), 0, n) ?
            0xff : 0x01);             // Activation signal
  else if ((t+n-1) % (3*r+3) == 0)
    return (inside ((t+n-1)/(3*r+3), 0, n) ?
            0xff : 0x01);             // Deactivation signal
  else
    return (0x01);                    // Don't care signal
}
int control2 (const int t)
{
  if ( t % (3*n) == 0)
    return (inside (t/(3*n), 0, n) ?
            0xff : 0x02);             // Activation signal
  if ( (t-2*n-2) % (3*n) == 0)
    return (inside ( (t-2*n-2)/(3*n), 0, n) ?
            0x00 : 0x02);             // Deactivation signal
  else
    return (0x02);                    // The don't care signal
}
int Ainput (const int t)
{
  int j = (t - 2 - (n-1)*(n-r)) % n;
  int i = (t - 2 - (n-1)*(n-r) - j)/n -j;

  if ((t == 2 + (n-1)*(n-r)+n*(i+j)+j)
       && inside (i, 0, n) && inside (j, 0, n))
    return A[i][j];
  else
    return 0;
}
int Binput (const int t)
{
  int j = (1-t) % (3 * (r+1));
  int i = (t + j - 1) / (3*(r+1));
  if ( (t == 1 + 3*(r+1)*(i)+2*j) &&
       inside (i, 0, n) && inside (j, 0, n))
    return B[i][j];
  else
    return 0;
}
void Coutput(const int t, const int val)
{
  int j = (array_size - t + 2) % (3*n);
  int i = (t - array_size + j - 2) / (3*n);
  if ( (t == array_size + 2 + 3*n*i + 2*j) &&
       inside (i, 1, n) && inside (j, 1, n))
    C[i][j] = val;
}
```

Figure 7.23: Transitive closure I/O functions.

Application	Lines
8-bit sorting	3
40-bit sorting	15
8-bit polynomial evaluation	34
24-bit polynomial evaluation	146
Sequence comparison	6
Sequence comparison, reset	7
Sequence comparison, affine deletion cost	24
Sequence comparison, affine, reset	27
Sequence comparison, subsequence location	6
Protein encoding region, 1 cell per location	16
Protein encoding region, 2 cells per location	12
Protein encoding region, reset	18
Protein encoding region, affine	52
Protein encoding region, affine, reset	54
Protein alignment, 2 cells per character	41
Protein alignment, 3 cells per character	28
Protein alignment, 4 cells per character	24
General transitive closure	10
Data compression	36
Data decompression	41

Table 7.2: B-SYS cell program lengths for various applications.

little modification.

7.3 Conclusions

Cell program lengths for several of the algorithms discussed in this chapter are presented in Table 7.2; the data compression and decompression lengths are based on Xu's B-SIM implementation of Storer and Reif's algorithm.[162,146,147] ASCII data compression is a combinatorial problem well suited to the Brown Systolic Array because only small amounts of memory and simple operations are required by the algorithm.

The software control of resources highlighted in the presentation of software fault detection methods (Section 5.3) is also applicable to general programming. In particular, extra processing elements can be readily used to perform protein encoding region searches and protein alignment computations more quickly. Programmable systolic arrays allow the user to decide the allocation of processing elements to a problem in addition to the algorithm to perform.

The Brown Systolic Arrays has proven its worth as a programmable systolic co-processor suitable for sequence comparison and other combinatorial applications. As the examples of this section have illustrated, a wide variety of sequence comparison algorithms can be implemented on the B-SYS system, a task greatly simplified by the New Systolic Language. Algorithms for simple sequence comparison, multiple sequence comparison, affine insertion and deletion costs for the introduction of gaps, alternate transcription, and protein alignment, as well as subsequence and longest common subsequence searches, can be quickly and efficiently implemented on the Brown Systolic Array. Unlike single-purpose systolic co-processors, B-SYS can be programmed for different cost metrics depending on the problem being solved, making it an invaluable tool for experimental sequence analysis. Unlike programmable hardware such as Splash, the generation and modification of B-SYS programs does not require a considerable amount of time and usually does not lower the systolic cell density. As was seen in the performance analysis of many-against-one sequence comparison, the Brown Systolic Array has the best chip-second resource value of all the systems considered, the one potential exception being BioSCAN, a currently unfinished system which will not be generally programmable.

Chapter 8

Conclusions

VERY large scale integration and wafer scale integration enable the placement of hundreds and thousands of processing elements on a single chip. Systolic arrays are one of the most efficient means of organizing these million-transistor chips; with their regular replication of simple cells, systolic arrays can outperform even the most powerful traditional supercomputer. This thesis has considered the design, implementation, and use of a programmable systolic array co-processor for combinatorial and other applications.

The design of any hardware system must evolve from a close examination of algorithms, hardware, and programming languages. The examination of systolic algorithms located the class of combinatorial problems as worthy of systolic solution. These problems require only simple operations and small amounts of memory, making them ideal for high-density VLSI solution. The study of systolic hardware found three approaches to systolic parallelism: the single-purpose array, the bit-serial SIMD array, and the systolic multiprocessor. Single-purpose arrays lack general programmability, restricting each one's use to a single hardwired algorithm. Bit-serial arrays generally have excessive memory for combinatorial problems and poorly designed systolic communication. Systolic multiprocessors are difficult to form into massively parallel systolic arrays because of their large processing elements, each with its own program store and instruction sequencer. As with the bit-serial arrays, systolic multiprocessors lack a sufficiently concise expression of systolic communication. In the review of systolic programming languages and design systems, it was discovered that no language is particularly suited for systolic co-processor programming. Systolic design systems provide a clean separation of systolic cell function and systolic data movement but are too abstract for general use. Low-level systolic languages force the user to consider irrelevant implementation details of the target machine. Thus, in systolic hardware and programming systems, there existed a gap which this thesis has filled for a specific class of systolic algorithms.

The analysis of existing systolic hardware led to the introduction of the Systolic Shared Register architecture. This architecture combines the efficiency of special-purpose systolic communication with the versatility of a programmable systolic array. The four defining attributes of the SSR architecture are:

- SIMD broadcast instructions for control

- regular interconnections for systolic data flow
- shared registers for communication and computation
- data streams for programming.

Broadcast instructions and regular interconnections simplify design and control, enabling the construction of high-density arrays. Shared registers elegantly implement systolic communication by separating functional units and register banks, breaking apart the traditional definition of a systolic cell. Systolic data streams ease the task of programming any systolic co-processor, combining the neatness of mathematical mapping with the flexibility of a real language. Although most work presented in this thesis concerns linear systolic arrays, it was shown that the SSR architecture provides efficient systolic communication for arrays of any topology.

Rather than leave the SSR design on paper, the Brown Systolic Array was designed, simulated, and fabricated in a 2 micron CMOS process. Its architecture reflects the targeted combinatorial applications, in particular the sequence comparison problems of molecular biologists. Thus, it has an 8-bit word and 16 registers per shared register bank. The 6.8 mm × 6.9 mm, 84-pin, 85 000-transistor chip worked on first fabrication; the success of such a complicated single-person VLSI design can be credited both to the simplicity of the SSR architecture and to the exhaustive and rigorous multi-level architecture and design simulation.

As with any VLSI implementation, several of the 24 fabricated chips had faults. The SSR architecture led to a fast test procedure for the B-SYS chips; functional units and register banks were quickly screened for errors before the construction of the 470-element B-SYS prototype system. This systolic co-processor can perform 470-character many-against-one sequence comparison over 80 times faster than its Intel 80386 host, and board modifications could magnify the speedup to 600. An aggressive redesign of the B-SYS chip could place 500 functional units and register banks on a single chip, producing an even more powerful systolic co-processor.

While fault testing is performed as a system is being constructed, fault detection takes place during array operation. For programmable systolic arrays, in particular those of the SSR design, software fault detection is the best general means of defending against transient faults. Fault-tolerant programs with little overhead can be automatically generated with the skewed replicated computation stream and interleaved replicated computation stream methods. Unlike hardware fault detection, these methods require no special circuitry and can be eliminated or strengthened as needed. Unlike algorithmic fault detection, these methods can be used with arbitrary programs.

After a system has been designed, tested, and constructed, thought must turn to its use. This thesis has proposed the New Systolic Language as a general framework for systolic programming. In addition to greatly simplifying B-SYS programming, NSL is a firm foundation for programming all types of systolic co-processors and simulators. Drawing on the merits and problems of existing systems, the New Systolic Language painlessly separates systolic cell function and macroscopic data movement to simplify the task of systolic programming.

With the New Systolic Language as a base, several B-SYS applications have been considered, including simple sequence comparison, sequence comparison with gap penalties, alternate transcription, protein alignment, and transitive closure. An analysis of 4-character sequence comparison on B-SYS and other supercomputers and co-processors lead to the conclusion that B-SYS does satisfy its design goals: it is a programmable systolic co-processor which greatly outperforms all other systems on target applications. Surprisingly, linear systolic arrays such as B-SYS can also perform traditionally 2-dimensional problems such as transitive closure. Although the transitive closure example was not the most efficient use of a linear systolic array, it illustrated the directness of coding a published algorithm in NSL.

Throughout this thesis, many directions for future research have been mentioned. The research area of most concern is the continued extension of the SSR architecture and NSL programming language to 2-dimensional systolic arrays. In particular, the implementation of hexagonal- and octagonal-mesh Systolic Shared Register machines is an important goal. Because of the limited I/O requirements of linear systolic arrays, shared registers were ideal for the Brown Systolic Array. The pin limitations of current technology will lead to interesting design tradeoffs when constructing a 2-dimensional VLSI Systolic Shared Register machine. In conjunction with this, an NSL implementation for both linear and mesh systolic arrays should be developed. Such work will bring up questions concerning the efficient mapping of algorithms to general classes of systolic co-processors as well as the concise expression of machine-independent systolic programs.

This thesis has developed a three-pronged approach to hardware design. Critical evaluations of existing algorithms, hardware, and programming languages led to the discovery of many inefficiencies and needless complexities. These were stripped away, producing the architectural requirements for combinatorial algorithms, the Systolic Shared Register architecture, the Brown Systolic Array, and the New Systolic Language. The combination of these three aspects of systolic co-processing forms a solid foundation for the creation of a unified systolic co-processing environment that integrates systolic co-processor algorithm development, programming, simulation, animation, and execution.

Appendix A

B-SIM User's Guide

THE Brown Systolic Array simulator is located in the /pro/cad/syssim directory. This appendix presents an overview of the simulator, though the best documentation of the simulator is in the example programs of the simulator directory.

As illustrated by Figure A.1, there are three major parts to B-SIM, or two without the Tango animation extensions. In the figure, boxes indicate programs or processes, italics represent control and data, and the remaining text refers to file inputs and outputs required by the simulator. The loopsym program processes command line options as well as two program files. The first program file (.init) is executed once at the start of the simulation, and the second (.instr) is executed as many times as is required by the command line arguments. In addition to the size of the array and the length of the simulation, the command line can specify several input files. These are referred to by the read field of a B-SIM instruction, as was seen in Figure 4.4 on page 54. Default values for input files can also be specified on the command line.

The actual simulator is run as a separate UNIX process by the loopsym program, much in the manner that a host machine controls a co-processor board. Inputs and instructions are passed to the simulator while outputs are retrieved from the simulator and sent to files by the loopsym program. Apart from the standard B-SYS instructions, there are several control directives which can be passed to the simulator (they are identified by 'impossible' flag addresses). Thus, comments can be passed to the simulator, and the simulator can be instructed to write the array's entire state to the sym.output file. This file may then be processed by the tf filter, which will produce TEX snapshots similar to those of Figure 4.5 on page 55. A .convert file can be used to provide encodings of register values as signed integers, unsigned integers, or special symbols.

The separation of simulation and simulation control will simplify the task of creating new front-ends for the simulator and the physical array. An NSL interface to the simulator is an obvious next step.

The Tango version of the simulator uses Field's message mechanism to pass simulation information to the Tango process.[134, 145] Several additional control directives can be used to shade specific registers or to temporarily halt the Tango animation. A slightly modified version of Tango (/pro/cad/syssim/src/tango/tango) must be used for B-SIM

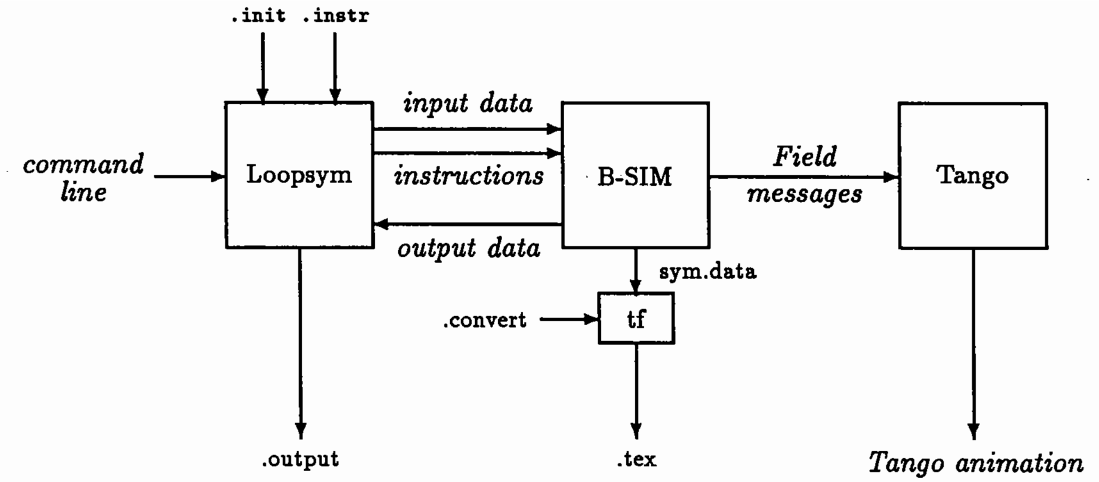


Figure A.1: The Brown Systolic Array simulator.

animations. The `syssim/tsort` example illustrates the use of Tango in conjunction with the simulator. The Tango program must be executed from the `tango` directory with `bsys` as an argument before using **make** to run the simulation. It is advisable to not run a window manager concurrently with the Tango animation.

Simulator execution is controlled with UNIX makefiles, examples of which may be found in the `syssim` example directories.

Appendix B

B-SYS User's Guide

THE Brown Systolic Array prototype board has been described in Section 5.2.2. This Appendix identifies some of the practical issues involved in executing programs on the B-SYS prototype system.

As mentioned, the prototype system is controlled by both the PC, which sends instructions and data to the board, and the HP 16520A pattern generator, which performs instruction clocking. To commence operation, the user must ensure that the pattern generator's D5 output pod is connected to header HA on the B-SYS board (Figure 5.2 on page 80), the pattern generator's W0 input on pod D1 is connected to pin P11 on the prototype board (the complemented select line for address 30C), and that the "IBM2" configuration is loaded by the logic analysis system. The pattern generator may then be placed in continuous execution mode. A short pattern generator program watches the address selection line and triggers a cycle of clock pulses whenever the third instruction word is accessed. The "IBM2" configuration allocates timing and state analyzer input pods for use with the prototype board, as can be investigated with the HP 16510A menus.

Programs associated with the B-SYS system are located in the `C:\BSYS` directory on the Intel 80386 PC. Test programs, written in a combination of assembly language and C, are located in the `\BSYS\TEST` directory, while several more complicated programs can be found in the `\BSYS\C` directory. The test routines, documented in the file `README.DOC`, follow the tests described in Section 5.2.

Typically, tests are invoked with the number of processing elements as an argument, and return no output if B-SYS passes the test. If B-SYS fails the test, the results are printed out for fault identification. Often, there are versions for performing tests from both sides of the array. For example, the aa55r and aa55l tests perform the InOut test of Section 5.2 from the right (east) and left (west) sides respectively. In addition to programs for performing the tests previously mentioned, a simple sorting routine is also available. It is suggested that the aa55 and sorting programs be run before any use of the B-SYS prototype board. The former will quickly identify the location of any chip seating errors, while the latter provides a more complete check on the system.

Programs in the style of Figure 7.2 on page 116 may be written using the `mopc.h` macro definitions (Figure 7.3 on page 117) located in the \BSYS\C directory. The fcompare program performs modulo sequence comparison with reset and compares performance with the host machine. If the results match, only the algorithm timings are displayed, otherwise the final column of the C dynamic programming matrix and the B-SYS outputs (which should correspond) are presented for analysis.

Although not as efficient as possible, the B-SYS prototype system enables the exploration of the inherent power of massively parallel programmable systolic arrays.

Appendix C

Sequence Comparison on the Connection Machine

THIS appendix describes the C* Connection Machine program used to generate the CM-2 data used in Section 7.1.2. For 100 × 100-element sequence comparison, the results of Lopresti's work were generated by performing each of the 100 complete comparisons on a single Connection Machine processing element, effectively turning the Connection Machine into a distributed processor.[58,111] Unless the database has as many sequences as the Connection Machine has processing elements, this algorithm will waste much of the vast processing power of the Connection Machine.

It is more efficient to configure the Connection Machine as a mesh of systolic pipelines. For the 100 × 100 case, this will not produce a factor of 100 speedup due to the overhead of passing data between processing elements, however the pipelined algorithm does run over 27 times faster than the distributed version on 16 K processing elements (0.17 s versus 4.7 s).

The driving routine for CM-2 sequence comparison is shown in Figure C.1. It selects a comparison routine based on the length of the sequences: numbers sufficient for the highest possible cost are used. (Since the final minimization over the database is done in the CM-2, true costs must be maintained in the end processing element of each pipeline, so modulo comparison is not used.)

The Connection Machine features virtual processing elements far beyond the physical capabilities of the machine, thus a large number of pipelines can be processed at once, as is indicated by the computation of the `pipes` variable in Figure C.1. After an appropriate mesh is specified, the database strings are loaded into a parallel variable and the target string is compared with the database.

As can be seen, the mesh specification is closely tied to the physical hardware: both mesh dimensions must be a power of two and the size of the mesh must be a power-of-two multiple of the number of physical processing elements. Experimentation with various mesh layouts found that excess processing elements should be assigned to excess pipelines: the length of the systolic pipelines should be kept as small as possible. Note that with at least 128 systolic pipes allocated, the program could compare 128 sequences

Length	Time (s)
10	0.023
20	0.041
50	0.090
100	0.17
200	0.46
500	2.32
1000	9.84

Table C.1: CM-2 100 sequence comparisons on 16384 processing elements (best time).

at about the same speed as 100 sequences. The complex constraints on grid size are a further indication that C* is a low-level programming language by the definitions of Section 2.3.

The sequence comparison cell program is shown in Figure C.2. The routine is divided into two parts, depending on whether or not the sequence has entered the pipeline. The algorithm closely follows that described previously. (The ?< operation performs minimization.) Experimental results for several sequence lengths are displayed in Table C.1.

There are a few improvements which could speed the algorithm, in particular for long sequences. First, four-bit characters could be used. Since the majority of execution time is taken up by the grid communication, reducing the amount of data flowing between virtual processing elements is vital. This optimization would be best performed in Paris, which allows the specification of arbitrary-length data. In the C* environment, the introduction of 4-bit characters would be tedious. For sequences with 128 or fewer characters, this would reduce grid communication by 25%.

When comparing long sequences, communication between processing elements is dominated by the transmission of cost data via 32-bit integers. Although it is important for the end processing elements to maintain true values at all times (for the final minimization across the entire database), modulo sequence comparison can still be used. If communication were restricted to 8-bit integers, reconstruction could be performed every sixty-third cell program iteration. Since local cost differences are at most 2, the values $d_{i-63,n}$ and $d_{i,n}$ cannot differ by more than 126, enabling modulo reconstruction of the true $d_{i,n}$ value from $d_{i-63,n}$ and an 8-bit difference. For long (32 768-character) sequences, such a change would cut communication by 75%, reduce the cell program length, and improve performance by perhaps fourfold.

```
/************************************************************************\
 *                                                                      *
 *       seq.cs                                                         *
 *       CM Sequence comparison main program                            *
 *                                                                      *
 *       Richard Hughey                                                 *
 *       16 March 1991                                                  *
 *                                                                      *
\************************************************************************/
#include <stdio.h>
#include <cscomm.h>
extern int load_dbase();
extern void compare_strings();
extern void compare_strings_char(), compare_strings_short();
void
compare  (int m,                /* Length of sequences */
          int n,                /* number of sequences */
          FILE *tgfile,         /* Target file   */
          FILE *dbfile)         /* Database file */
{
  int pipes, plen, i;           /* Number and length of systolic pipes */
  int best_match, best_cost;
  char *tgt;
  void (*cfunction)();
  shape [][]ManyPipes;

  if (m < 128)
    cfunction = compare_strings_char;
  else if (m < 32768)
    cfunction = compare_strings_short;
  else
    cfunction = compare_strings;

  /* Must maintain powers of 2 for plen and pipes, and */
  /* their product must be a multiple of the machine size. */
  for (plen = 2;  (plen < (m+1)); plen <<= 1);
  for (i = 1; (pipes = i*positionsof(physical)/plen) < n; i<<=1);

  tgt = (char *) malloc ((m+1) * sizeof (char));
  fgets (tgt, m+1, tgfile);

  /* communication is much greater across the second axis */
  ManyPipes = allocate_detailed_shape (&ManyPipes, 2,
       pipes, 1, CM_news_order, 0, 0, plen, 100, CM_news_order, 0, 0);

  with (ManyPipes) {
    everywhere {
      char:current string2;
      load_dbase (dbfile, m, n, &string2);
      compare_strings (tgt, m, n, &string2, &best_match, &best_cost);
    }
  }
```

Figure C.1: CM sequence comparison driving routine.

```
void compare_strings (char *tgt, int m, int n,
                      char:current *string2,
                      int *best_match, int *best_cost)
{
  int i;
  unsigned int:current cost2, cost1, lcost1, lcost2, fill;
  unsigned int:current *lastcost, *newcost, *leftcost, *diagcost, *temp;
  char:current str1, cfill; char:current *string1;

  string1 = &str1;
  lastcost = &cost1; newcost = &cost2;
  leftcost = &lcost1; diagcost = &lcost2;

    /* Initial conditions */
  *string1 = 0;  *newcost = 1; *lastcost = *leftcost = 0;
  for (i = 0 ; i < m ; i++) {
      /* Step input costs */
    swap (diagcost, leftcost, temp);
    swap (lastcost, newcost, temp);

      /* shift costs and characters */
    fill = i+1;
    to_grid_dim (leftcost, *lastcost, &fill, 1, 1);
    cfill = tgt[i];
    to_grid_dim (string1, *string1, &cfill, 1, 1);
      /* Take min of left and last and add 1 for non-match cost. */
    *newcost = (*leftcost <? *lastcost);
    *newcost++;
      /* Use diagcost if characters match */
    where (*string1 == *string2)
      newcost = *diagcost;
  }
  for (i = 0 ; i < m-1 ; i++) {
    swap (diagcost, leftcost, temp);
    swap (lastcost, newcost, temp);
      /* shift costs and characters in, ignore first pe */
    to_grid_dim (leftcost, *lastcost, &fill, 1, 1);
    to_grid_dim (string1, *string1, &cfill, 1, 1);
    *newcost = (*leftcost <? *lastcost);
    *newcost++;
    where (*string1 == *string2)
      newcost = *diagcost;
  }
    /* lastpe newcost now has sequence comparison results. */
    /* locate minimum of all pipes. */
  where ((pcoord(1) == (m-1)) && (pcoord(0) < n)) {
    *best_cost = <?= *newcost;
    where (*newcost == *best_cost) {
      best_match = (int) pcoord (0);
    }
  }
}
```

Figure C.2: CM sequence comparison cell program.

Bibliography

[1] B. Ackland, N. Weste, and D. J. Burr, "An integrated multiprocessing array for time warp pattern matching," in *Proc. Int. Symp. Computer Architecture*, pp. 197–215, IEEE, 1981.

[2] B. D. Ackland, "Dynamic time warp processor," in *Principles of CMOS Design*, pp. 384–407, Reading, MA: Addison-Wesley Publishing Company, 1985.

[3] A. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley Publishing Company, 1974.

[4] A. V. Aho and T. G. Peterson, "A minimum distance error-correcting parser for context-free languages," *SIAM J. Computing*, vol. 1, pp. 305–312, Dec. 1972.

[5] M. Annaratone, *Digital CMOS Circuit Design*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 1986.

[6] M. Annaratone *et al.*, "Warp architecture and implementation," in *Proc. Int. Symp. Computer Architecture*, pp. 346–356, IEEE, 1986.

[7] M. Annaratone *et al.*, "Architecture of Warp," in *COMPCON Spring '87*, pp. 264–267, IEEE Computer Society, 1987.

[8] M. Annaratone *et al.*, "The Warp computer: Architecture, implementation and performance," *IEEE Trans. Computers*, vol. C-36, pp. 1523–1537, Dec. 1987.

[9] J. Annevelink and P. Dewilde, "HIFI: A functional design system for VLSI processing arrays," in *Proc. First Int. Conf. Systolic Arrays* (K. Bromley, S. Y. Kung, and E. Swartzlander, eds.), pp. 413–452, IEEE Computer Society, May 1988.

[10] D. K. Arvind, I. N. Robinson, and I. N. Parker, "A VLSI chip for real-time image processing," in *Proc. Int. Symp. Circuits and Systems*, vol. 1, pp. 405–408, IEEE, May 1983.

[11] M. J. Atallah and S. R. Kosaraju, "Graph problems on a mesh-connected processor array," in *Symp. Theory of Computation*, pp. 345–353, ACM, 1982.

[12] V. Balasubramanian and P. Banerjee, "Compiler-assisted synthesis of algorithm-based checking in multiprocessors," *IEEE Trans. Computers*, vol. 39, pp. 436–446, Apr. 1990.

[13] K. E. Batcher, "Design of a massively parallel processor," *IEEE Trans. Computers*, pp. 836–840, Sept. 1980.

[14] D. Beatty *et al.*, *User's Guide to COSMOS version 1.2.* Carnegie-Mellon University, Oct. 1988.

[15] M. J. Bishop and C. J. Rawlings, eds., *Nucleic acid and protein sequence analysis.* Oxford, England: IRL Press, 1987.

[16] J. Blackmer, G. Frank, and P. Kuekes, "A 200 million operations per second (MOPS) systolic processor," in *Real-Time Signal Processing IV (Proc. SPIE)*, vol. 298, pp. 10–18, 1982.

[17] D. W. Blevins *et al.*, "BLITZEN: A highly integrated massively parallel machine," *J. Parallel Distributed Computing*, vol. 8, pp. 150–160, Feb. 1990.

[18] S. Borkar *et al.*, "iWarp: An integrated solution to high-speed parallel computing," in *Proc. Supercomputing '88*, pp. 330–339, IEEE, Nov. 1988.

[19] R. P. Brent and H. T. Kung, "Systolic VLSI architectures for polynomial GCD computation," *IEEE Trans. Computers*, vol. 33, no. 8, pp. 731–736, 1984.

[20] K. Bromley, S. Y. Kung, and E. Swartzlander, eds., *Proc. First Int. Conf. Systolic Arrays*, IEEE Computer Society, May 1988.

[21] B. Bruegge, C. Chang, R. Cohn, T. Gross, M. Lam, P. Lieu, A. Noaman, and D. Yam, "Programming Warp," in *COMPCON Spring '87*, pp. 268–271, IEEE Computer Society, 1987.

[22] C. R. Cantor, "Orchestrating the Humane Genome Project," *Science*, vol. 248, pp. 49–51, 6 Apr. 1990.

[23] P. R. Capello and K. Steiglitz, "Digital signal processing applications of systolic algorithms," in *VLSI Systems and Computation* (H. T. Kung, B. Sproull, and G. Steele, eds.), pp. 245–254, Rockville, MD: Computer Science Press, 1981.

[24] J. H. Chang, O. H. Ibarra, and M. A. Palis, "Parallel parsing on a one-way array of finite-state machines," *IEEE Trans. Computers*, vol. 36, pp. 64–75, Jan. 1987.

[25] B. Chazelle, "Computational geometry on a systolic chip," *IEEE Trans. Computers*, vol. 33, pp. 774–785, Sept. 1984.

[26] E. A. Cheever, G. C. Overton, and D. B. Searls, "Fast fourier transform-based correlation of DNA sequences using complex plane encoding," Tech. Rep. PRC-KSC-8907, Unisys, Paoli Research Center, Paoli, PA, Nov. 1989. To appear in CABIOS.

[27] Y.-H. Choi, S. H. Han, and M. Malek, "Fault diognosis of reconfigurable arrays," in *Proc. Int. Conf. Computer Design*, pp. 451–455, IEEE, 1984.

[28] N. H. Christ and A. E. Terrano, "A micro-based supercomputer," *Byte*, pp. 145–160, Apr. 1986.

[29] "Special issue on systolic arrays," *Computer*, vol. 20, July 1987.

[30] N. G. Core, E. W. Edmiston, J. H. Saltz, and R. M. Smith, "Supercomputers and biological sequence comparison algorithms," *Computers and Biomedical Research*, vol. 22, pp. 497–515, 1989.

[31] R. J. Cosentino, "Fault tolerance in a systolic residue arithmetic processor array," *IEEE Trans. Computers*, vol. 37, pp. 886–890, July 1988.

[32] M. O. Dayhoff, R. M. Schwartz, and B. C. Orcutt, "A model of evolutionary change in proteins," in *Atlas of Protein Sequence and Structure*, ch. 22, pp. 345–358, Washington, D. C.: National Biomedical Research Foundation, 1978.

[33] C. DeLisi, "Computers in molecular biology: Current applications and emerging trends," *Science*, vol. 246, pp. 47–51, 6 Apr. 1988.

[34] C. DeLisi, "The Human Genome Project," *American Scientist*, vol. 76, pp. 488–493, Sept. 1988.

[35] W. D. Dettloff, R. K. Singh, C. T. White, and B. W. Erickson, "A 50MHz 1.5M transistor ASIC for biosequence analysis," in *Proc. Int. Symp. Solid State Circuits Digest of Technical Papers*, Feb. 1991.

[36] J. Deutch *et al.*, "Performance of Warp on the DARPA image understanding architecture benchmarks," in *Parallel Processing for Computer Vision and Display* (P. M. Dew, R. A. Earnshaw, and T. R. Heywood, eds.), ch. 8, pp. 119–135, Reading, MA: Addison-Wesley Publishing Company, 1989.

[37] P. M. Dew, R. A. Earnshaw, and T. R. Heywood, eds., *Parallel Processing for Computer Vision and Display.* Reading, MA: Addison-Wesley Publishing Company, 1989.

[38] V. V. Dongen, "Quasi-regular arrays: definition and design methodology," in *Systolic Array Processors* (J. McCanny, J. McWhirter, and J. Earl Swartzlander, eds.), pp. 126–135, Englewood Cliffs, NJ: Prentice-Hall, 1989.

[39] K. A. Doshi and P. J. Varman, "Optimal graph algorithms on a fixed-size linear array," *IEEE Trans. Computers*, vol. 36, pp. 460–470, Apr. 1987.

[40] A. El-Amawy, "A systolic architecture for fast dense matrix inversion," *IEEE Trans. Computers*, vol. 38, no. 3, pp. 449–455, 1989.

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

[42] B. R. Engstrom and P. R. Capello, "The SDEF systolic programming system," in *Concurrent Computations* (S. K. Tewksbury, B. W. Dickinson, and S. C. Schwartz, eds.), ch. 15, pp. 263–301, New York: Plenum Press, 1988.

[43] Z. Fang, X. Li, and L. M. Ni, "On the communication complexity of 2-D convolution on array processors," *IEEE Trans. Computers*, vol. 38, no. 2, pp. 184–194, 1989.

[44] R. Farber *et al.*, "Determination of eucaryotic protein coding regions using neural networks and information theory." Private communication, Oct. 1990.

[45] A. L. Fisher, "Systolic algorithms for running order statistics in signal and image processing," in *VLSI Systems and Computation* (H. T. Kung, B. Sproull, and G. Steele, eds.), pp. 265–272, Rockville, MD: Computer Science Press, 1981.

[46] A. L. Fisher and P. T. Highnam, "Computing the Hough transform on a scan line array processor," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 11, no. 3, pp. 262–265, 1989.

[47] A. L. Fisher, P. T. Highnam, and T. E. Rockoff, "Architecture of a VLSI SIMD processing element," in *Proc. Int. Conf. Computer Design*, pp. 324–327, IEEE, Oct. 1987.

[48] A. L. Fisher, H. T. Kung, *et al.*, "Design of the PSC: A programmable systolic chip," in *Third CALTECH Conference on VLSI* (R. Bryant, ed.), pp. 287–302, Rockville, MD: Computer Science Press, 1983.

[49] P. M. Flanders, D. J. Hunt, S. F. Reddaway, and D. Parkinson, "Efficient high speed computing with the Distributed Array Processor," in *High Speed Computer and Algorithm Organization* (D. J. Kuck, D. H. Lawrie, and A. H. Sameh, eds.), pp. 113–128, San Diego, CA: Academic Press, 1977.

[50] M. J. Foster and H. T. Kung, "The design of special-purpose VLSI chips," *Computer*, pp. 26–40, Jan. 1980.

[51] D. E. Foulser and R. Schreiber, "The Saxpy Matrix-1: A general purpose systolic computer," *Computer*, vol. 20, pp. 35–43, July 1987.

[52] T. Fountain, *Processor Arrays: Architectures and Applications.* San Diego, CA: Academic Press, 1987.

[53] T. J. Fountain, "A survey of bit-serial array processor circuits," in *Computing Structures of Image Processing* (M. J. B. Duff, ed.), ch. 1, pp. 1–14, San Diego, CA: Academic Press, 1983.

[54] T. J. Fountain, K. N. Mathews, and M. J. B. Duff, "The CLIP7A image processor," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 10, pp. 310–319, Mar. 1988.

[55] P. Frison, D. Lavenier, H. Leverge, and P. Quinton, "MICSMACS: A VLSI programmable systolic architecture," in *Systolic Array Processors* (J. McCanny, J. McWhirter, and J. Earl Swartzlander, eds.), pp. 146–154, Englewood Cliffs, NJ: Prentice-Hall, 1989.

[56] P. Gachet, B. Joinnault, and P. Quinton, "Synthesizing systolic arrays using DIASTOL," in *Systolic Arrays* (W. Moore, A. McCabe, and R. Urquhart, eds.), pp. 25–37, Boston, MA: Adam Hilger, 1987.

[57] F. M. F. Gaston and G. W. Irwin, "A systolic square root information Kalman filter," in *Proc. First Int. Conf. Systolic Arrays* (K. Bromley, S. Y. Kung, and E. Swartzlander, eds.), pp. 643–652, IEEE Computer Society, May 1988.

[58] M. Gokhale *et al.*, "Building and using a highly parallel programmable logic array," *Computer*, vol. 24, pp. 81–89, Jan. 1991.

[59] T. Gross and M. S. Lam, "Compilation for a high-performance systolic array," in *Proc. 1986 Symp. Compiler Construction*, pp. 27–38, ACM SIGPLAN, 1986.

[60] C. Guerra and T. Kanade, "A systolic algorithm for stereo matching," in *VLSI: Algorithms and Architectures* (P. Bertolazzi and F. Luccio, eds.), pp. 103–112, Amsterdam, Holland: North-Holland, 1985.

[61] L. J. Guibas, H. T. Kung, and C. D. Thompson, "Direct VLSI impementation of combinatorial algorithms," in *CALTECH Conference on VLSI*, pp. 509–525, 1979.

[62] A. Haug and R. Graybill, "The Martin Marietta Advance Systolic Array Processor," in *Proc. Symp. the Frontiers of Massively Parallel Computation* (R. Mills, ed.), pp. 367–372, IEEE, Oct. 1988.

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

[64] C. A. R. Hoare, "Communicating sequential processes," *Communications ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[65] M. Homewood, D. May, D. Shepherd, and R. Shepherd, "The IMS T800 Transputer," *IEEE Micro*, pp. 10–26, Oct. 1987.

[66] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Computers*, vol. c-33, pp. 318–528, June 1984.

[67] R. Hughey and D. Lopresti, "An architecture for programmable systolic arrays," Tech. Rep. CS-89-08, Department of Computer Science, Brown University, Providence, RI, Feb. 1989.

[68] R. Hughey and D. P. Lopresti, "Location of protein encodings in nucleic acids," Tech. Rep. CS-87-09, Department of Computer Science, Brown University, Providence, RI, May 1987.

[69] R. Hughey and D. P. Lopresti, "Architecure of a programmable systolic array," in *Proc. First Int. Conf. Systolic Arrays* (K. Bromley, S. Y. Kung, and E. Swartzlander, eds.), pp. 41–50, IEEE Computer Society, May 1988.

[70] R. Hughey and D. P. Lopresti, "A software approach to fault detection on programmable systolic arrays," in *Proc. Symp. Parallel and Distributed Processing* (B. Shirazi and H. Sudborough, eds.), pp. 523–526, IEEE Computer Society, Dec. 1990.

[71] O. H. Ibarra *et al.*, "Systolic algorithms for some scheduling and graph problems," in *Proc. First Int. Conf. Systolic Arrays* (K. Bromley, S. Y. Kung, and E. Swartzlander, eds.), pp. 247–256, IEEE Computer Society, May 1988.

[72] IEE, *Proc. Int. Symp. Computer Architecture and Digital Signal Processing*, (Hong Kong), Oct. 1989.

[73] N. Jagadish, J. M. Kumar, and L. M. Patnaik, "An efficient scheme for interprocessor communication using dual-ported RAMs," *IEEE Micro*, pp. 10–19, Oct. 1989.

[74] C. S. Jeong and D. T. Lee, "Parallel convex hull algorithms in 2D and 3D on mesh-connected computers," in *Parallel Processing for Computer Vision and Display* (P. M. Dew, R. A. Earnshaw, and T. R. Heywood, eds.), ch. 4, pp. 64–76, Reading, MA: Addison-Wesley Publishing Company, 1989.

[75] H. F. Jordan, "HEP architecture, programming and performance," in *Parallel MIMD Computation: HEP Supercomputer and Its Applications* (J. S. Kowalik, ed.), ch. 1.1, pp. 1–39, Cambridge, MA: MIT Press, 1985.

[76] D. C. Kar, V. V. B. Rao, and C.-S. Poon, "A high-performance VLSI systolic array for computing projection operators," in *Proc. Int. Symp. Computer Architecture and Digital Signal Processing*, (Hong Kong), pp. 75–79, IEE, Oct. 1989.

[77] S. Karline and S. F. Altschul, "Methods for assessing the statistical significance of melecular sequence features by using general scoring schemes," *Proc. Natl. Acad. Sci. USA*, vol. 87, pp. 2264–2268, Mar. 1990.

[78] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *J. ACM*, vol. 14, pp. 563–590, July 1967.

[79] T. Kilburn, D. B. G. Edwards, and D. Aspinall, "A parallel arithemetic unit using a saturated-transistor fast-carry circuit," *Proc. IEE, Part B*, vol. 107, pp. 573–584, Nov. 1960.

[80] J. H. Kim, "A fault-tolerant systolic array design using TMR method," in *Proc. Int. Conf. Computer Design*, pp. 769–773, IEEE, 1985.

[81] S. C. Knowles and J. G. McWhirter, "An improved bit-level systolic architecture for IIR filtering," in *Systolic Array Processors* (J. McCanny, J. McWhirter, and J. Earl Swartzlander, eds.), pp. 205–214, Englewood Cliffs, NJ: Prentice-Hall, 1989.

[82] S. C. Knowles, J. G. McWhirter, and R. A. Evans, "A high-performance bit-level systolic array for IIR filtering," in *Proc. Int. Symp. Computer Architecture and Digital Signal Processing*, (Hong Kong), pp. 553–557, IEE, Oct. 1989.

[83] T. Kondo, T. Nakashima, M. Aoki, and T. Sudo, "An LSI adaptive array processor," *IEEE J. Solid State Circuits*, vol. sc-18, pp. 147–156, Apr. 1983.

[84] J. S. Kowalik, ed., *Parallel MIMD Computation: HEP Supercomputer and Its Applications.* Cambridge, MA: MIT Press, 1985.

[85] J. B. Kruskal and D. Sankoff, "An anthology of algorithms and concepts for sequence comparison," in *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, pp. 265–310, Reading, MA: Addison-Wesley Publishing Company, 1983.

[86] A. V. Kulkarni and D. W. L. Yen, "Systolic processing and an implementation for signal and image processing," *IEEE Trans. Computers*, vol. 31, pp. 1000–1009, Oct. 1982.

[87] M. Kunde *et al.*, "The Instruction Systolic Array and its relation to other models of parallel computers," in *Parallel Computing '85* (M. Feilmeier, G. Joubert, and U. Schendel, eds.), pp. 491–497, Amsterdam, Holland: North-Holland, 1986.

[88] H. T. Kung, "Use of VLSI in algebraic computation: some suggestions," in *Proc. Symp. Symbolic and Algebraic Computation*, pp. 218–222, ACM, 1981.

[89] H. T. Kung, "Why systolic architectures?," *Computer*, pp. 37–46, Jan. 1982.

[90] H. T. Kung, "Deadlock avoidance for systolic communication," *Journal of Complexity*, vol. 4, pp. 87–105, 1988.

[91] H. T. Kung, "Systolic communication," in *Proc. First Int. Conf. Systolic Arrays* (K. Bromley, S. Y. Kung, and E. Swartzlander, eds.), pp. 695–703, IEEE Computer Society, May 1988.

[92] H. T. Kung and P. L. Lehman, "Systolic (VLSI) arrays for relational database operations," in *Proc. Int. Conf. Management of Data (SIGMOD)* (P. P. Chen and R. C. Sprowls, eds.), pp. 105–116, ACM, May 1980.

[93] H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in *Sparse Matrix Proceedings* (I. S. Duff and G. W. Stewart, eds.), pp. 256–282, SIAM, 1978.

[94] H. T. Kung and C. E. Leiserson, "Algorithms for VLSI processor arrays," in *Introduction to VLSI Systems*, ch. 8.3, pp. 271–292, Reading, MA: Addison-Wesley Publishing Company, 1980.

[95] H. T. Kung, B. Sproull, and G. Steele, eds., *VLSI Systems and Computation.* Rockville, MD: Computer Science Press, 1981.

[96] S. Y. Kung, "VLSI array processors," in *Systolic Arrays* (W. Moore, A. McCabe, and R. Urquhart, eds.), pp. 7–25, Boston, MA: Adam Hilger, 1987.

[97] S. Y. Kung, *VLSI Array Processors.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

[98] S. Y. Kung, "Array compiler design for VLSI/WSI systems," in *Systolic Array Processors* (J. McCanny, J. McWhirter, and J. Earl Swartzlander, eds.), app. 2, Englewood Cliffs, NJ: Prentice-Hall, 1989.

[99] M. S. Lam, *A Systolic Array Optimizing Compiler.* Dordrecht, The Netherlands: Kluwer Academic Publishers, 1989.

[100] H.-W. Lang, "The Instruction Systolic Array — a parallel architecture for VLSI," *Integration, the VLSI Journal*, vol. 4, no. 1, pp. 65–74, 1986.

[101] P. Lee and Z. Kedem, "On high-speed computing with a programmable linear array," in *Proc. Supercomputing '88*, pp. 425–432, IEEE, Nov. 1988.

[102] P. Lee and Z. Kedem, "Synthesizing linear array algorithms from nested FOR loop algorithms," *IEEE Trans. Computers*, vol. 37, pp. 1578–1598, Dec. 1988.

[103] P. L. Lehman, "A systolic (VLSI) array for processing simple relational queries," in *VLSI Systems and Computation* (H. T. Kung, B. Sproull, and G. Steele, eds.), pp. 285–295, Rockville, MD: Computer Science Press, 1981.

[104] C. E. Leiserson, *Area-Efficient VLSI Computation.* Cambridge, MA: MIT Press, 1983.

[105] A. M. Lesk, ed., *Computational Molecular Biology.* Oxford, England: Oxford University Press, 1988.

[106] H. F. Li, D. Pao, and R. Jayakumar, "Systolic arrays: present state and issues," in *Proc. Int. Symp. Computer Architecture and Digital Signal Processing*, (Hong Kong), pp. 69–74, IEE, Oct. 1989.

[107] F.-C. Lin and K. Chen, "On the design of a unidirectional systolic array for key-enumeration," *IEEE Trans. Computers*, vol. 39, pp. 266–269, Feb. 1990.

[108] 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. 363–376, Boston, MA: Adam Hilger, 1987.

[109] D. P. Lopresti, *Discounts for Dynamic Programming with Applications in VLSI Processor Arrays.* PhD thesis, Princeton University, Princeton, N.J., 1987.

[110] D. P. Lopresti, "P-NAC: A systolic array for comparing nucleic acid sequences," *Computer*, vol. 20, pp. 98–99, July 1987.

[111] D. P. Lopresti, "Sequence comparison on commercial supercomputers," Tech. Rep. SRC-TR-89-010, Supercomputing Research Center, Bowie, MD, Oct. 1989.

[112] D. P. Lopresti. Personal communication, Jan. 1991.

[113] W. Luk and G. Jones, "The derivation of regular synchronous circuits," in *Proc. First Int. Conf. Systolic Arrays* (K. Bromley, S. Y. Kung, and E. Swartzlander, eds.), pp. 305–314, IEEE Computer Society, May 1988.

[114] G. Lyon, "Syntax-directed least-errors analysis for context free languages: A practical approach," *Communications ACM*, vol. 17, pp. 3–14, Jan. 1974.

[115] S. Manohar, "Systolic architectures: A critical survey," Tech. Rep. CS-86-05, Department of Computer Science, Brown University, Providence, RI, Mar. 1986.

[116] W. J. Masek and M. S. Paterson, "How to compute string-edit distances quickly," in *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, pp. 337–349, Reading, MA: Addison-Wesley Publishing Company, 1983.

[117] D. May and R. Taylor, "OCCAM — an overview," *Microprocessors and Microsystems*, vol. 8, pp. 73–79, Mar. 1984.

[118] R. N. Mayo *et al.*, "1990 DECWRL/Livermore Magic Release," Research Report 90/7, Digital Western Research Laboratory, Palo Alto, CA, Sept. 1990.

[119] J. McCanny, J. McWhirter, and J. Earl Swartzlander, eds., *Systolic Array Processors*. Englewood Cliffs, NJ: Prentice-Hall, 1989.

[120] C. A. Mead and L. A. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley Publishing Company, 1980.

[121] D. Moldovan, "On the analysis and synthesis of VLSI algorithms," *IEEE Trans. Computers*, vol. c-31, pp. 1121–1126, Nov. 1982.

[122] D. I. Moldovan, "On the design of algorithms for VLSI systolic arrays," *Proc. IEEE*, vol. 71, pp. 113–120, Jan. 1983.

[123] D. I. Moldovan and J. A. B. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE Trans. Computers*, vol. c-35, pp. 1–12, Jan. 1986.

[124] W. Moore, A. McCabe, and R. Urquhart, eds., *Systolic Arrays*. Boston, MA: Adam Hilger, 1987.

[125] R. E. Morley and T. J. Sullivan, "A massively parallel systolic array processor system," in *Proc. First Int. Conf. Systolic Arrays* (K. Bromley, S. Y. Kung, and E. Swartzlander, eds.), pp. 217–226, IEEE Computer Society, May 1988.

[126] J. G. Nash and C. Petrozolin, "VLSI implementation of a linear systolic array," in *Proc. Int. Conf. Acoustics, Speech, and Signal Processing*, pp. 1392–1395, IEEE, 1985.

[127] NCR Corp., *Geometric Arithmetic Parallel Processor Data Sheet, part number NCR45CG72*, 1987.

[128] D. P. O'Leary, "Systolic arrays for matrix transpose and other reorderings," *IEEE Trans. Computers*, vol. 36, pp. 117–122, Jan. 1987.

[129] J. Osterhout, "Using Crystal for timing analysis," in *1986 VLSI Tools: Still More Work by the Original Artists*, no. UCB/CSD 86/272, Dec. 1986.

[130] J. H. Patel and L. Y. Fung, "Concurrent error detection in ALU's by recomputing with shifted operands," *IEEE Trans. Computers*, vol. c-31, pp. 589–595, July 1982.

[131] H. Peltola, H. Söderland, and E. Ukkonen, "Algorithms for the search of amino acid patterns in nucleic acid sequences," *Nucleic Acids Research*, vol. 14, no. 1, pp. 99–107, 1986.

[132] J. L. Potter, ed., *The Massively Parallel Processor.* Cambridge, MA: MIT Press, 1985.

[133] I. V. Ramakrishnan and P. J. Varman, "Modular matrix multiplication on a linear array," *IEEE Trans. Computers*, vol. c-33, pp. 952–958, Nov. 1984.

[134] S. P. Reiss, "Connecting tools using message passing in the FIELD program development environment," *IEEE Software*, July 1990. Also available as Brown University Computer Science Department Technical Report CS-88-18, "Integration Mechanisms in the FIELD Environment," 1988.

[135] D. Sankoff and J. B. Kruskal, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Reading, MA: Addison-Wesley Publishing Company, 1983.

[136] L. A. Schmitt and S. S. Wilson, "The AIS-5000 parallel processor," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 10, pp. 320–330, Mar. 1988.

[137] R. M. Schwartz and M. O. Dayhoff, "Matrices for detecting distant relationships," in *Atlas of Protein Sequence and Structure*, ch. 23, pp. 353–358, Washington, D. C.: National Biomedical Research Foundation, 1978.

[138] R. Sedgewick, *Algorithms.* Reading, MA: Addison-Wesley Publishing Company, 2 ed., 1988.

[139] G. M. Shaw. Personal communication, 1987.

[140] T. Shepherd and J. McWhirter, "A systolic array for linearly constrained least-squares optimisation," in *Systolic Arrays* (W. Moore, A. McCabe, and R. Urquhart, eds.), ch. 2.6, pp. 151–159, Boston, MA: Adam Hilger, 1987.

[141] L. A. Shombert and D. P. Siewiorek, "Using redundancy for concurrent testing and repairing of systolic arrays," in *Int. Symp. Fault-Tolerant Computing, Digest of Papers*, pp. 244–249, IEEE, July 1987.

[142] R. K. Singh, V. L. Chi, and T. White. Personal communication, Oct. 1990.

[143] J. Smallbone, "Programming high performance graphics on the DAP," in *Parallel Processing for Computer Vision and Display* (P. M. Dew, R. A. Earnshaw, and T. R. Heywood, eds.), ch. 23, pp. 321–328, Reading, MA: Addison-Wesley Publishing Company, 1989.

[144] L. Snyder, "Hearts: A dialect of the Poker programming environment specialised to systolic computation," in *Systolic Arrays* (W. Moore, A. McCabe, and R. Urquhart, eds.), pp. 71–80, Boston, MA: Adam Hilger, 1987.

[145] J. T. Stasko, "TANGO: A framework and system for algorithm animation," *Computer*, vol. 23, pp. 27–39, Sept. 1990.

[146] J. A. Storer, *Data Compression.* Rockville, MD: Computer Science Press, 1988.

[147] J. A. Storer and J. H. Reif, "A parallel architecture for high speed data compression." Extended abstract, personal communication, Jan. 1990.

[148] E. Swartzlander, "Systolic FFT processors," in *Systolic Arrays* (W. Moore, A. McCabe, and R. Urquhart, eds.), ch. 2.4, pp. 133–140, Boston, MA: Adam Hilger, 1987.

[149] Thinking Machines Corporation, Cambridge, MA, *Paris Release Notes (version 5.1)*, June 1989.

[150] C. D. Thompson, "The VLSI complexity of sorting," in *VLSI Systems and Computation* (H. T. Kung, B. Sproull, and G. Steele, eds.), pp. 108–118, Rockville, MD: Computer Science Press, 1981.

[151] R. R. Troutman, *Latchup in CMOS Technology.* Dordrecht, The Netherlands: Kluwer Academic Publishers, 1986.

[152] L. W. Tucker and G. G. Robertson, "Architecture and applications of the Connection Machine," *Computer*, vol. 21, pp. 26–38, Aug. 1988.

[153] P. J. Varman and I. V. Ramakrishnan, "Dynamic programming and transitive closure on linear pipelines," in *Proc. Int. Conf. Parallel Processing* (R. M. Keller, ed.), pp. 359–364, IEEE, Aug. 1984.

[154] G. von Heijne, *Sequence Analaysis in Molecular Biology.* San Diego, CA: Academic Press, 1987.

[155] R. Wagner, "Formal-language error correction," in *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, pp. 331–336, Reading, MA: Addison-Wesley Publishing Company, 1983.

[156] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM*, vol. 21, pp. 168–173, Jan. 1974.

[157] J. D. Watson, "The Human Genome Project: Past, present, and future," *Science*, vol. 248, pp. 44–48, 6 Apr. 1990.

[158] N. H. E. Weste and K. Eshraghian, *Principles of CMOS Design.* Reading, MA: Addison-Wesley Publishing Company, 1985.

[159] T. White, "Personal communication," 1991.

[160] T. Williams, "Massively parallel processing array spits out 30,000 Mips," *Computer Design*, pp. 45–46, Oct. 1989.

[161] Xilinx, Inc., San Jose, CA, *The Programmable Gate Array Design Handbook*, 1986.

[162] J. Xu, "A parallel implementation of on-line data compression." Personal communication, Jan. 1991.

[163] Y. Yaacoby and P. R. Capello, "Scheduling a system of affine recurrence equations onto a systolic array," in *Proc. First Int. Conf. Systolic Arrays* (K. Bromley, S. Y. Kung, and E. Swartzlander, eds.), pp. 373–382, IEEE Computer Society, May 1988.

[164] C. N. Zhang, H. L. Martin, and D. Y. Y. Yun, "Parallel algorithms and systolic array designs for RSA cryptosystem," in *Proc. First Int. Conf. Systolic Arrays* (K. Bromley, S. Y. Kung, and E. Swartzlander, eds.), pp. 341–350, IEEE Computer Society, May 1988.

[165] "Understanding our genetic inheritance — the U.S. Human Genome Project: The first five years," Tech. Rep. DOE/ER-0452P, U.S. Department of Health and Human Services and the U.S. Department of Energy, 1990.