REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

nated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and
rmation. Send comments regarding this burden estimate or any other subject of this collection of information, including suggestions
Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to
ment and Budget, Washington, DC 20603.

AD-A244 057

2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
1 Oct 91	Final 1 Jul 91 - 1 Oct 91

Novel Optical Computer Architecture Utilizing Reconfigurable Interconnects - Final Report

5. FUNDING NUMBERS

F49620-91-C-0055

6. AUTHOR(S)

Miles Murdocca, Apostolos Gerasoulis, and Saul Levy

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Technical Imaging Services (TIS Incorporated)
380 Farmingdale Road
Jackson, NJ 08527

8. PERFORMING ORGANIZATION REPORT NUMBER

TR-100191.01

DTIC
ELECTE
DEC 30 1991
S D
D

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Office of Scientific Research
Building 410
Bolling AFB DC 20332-6448

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

SDI
1602/01

11. SUPPLEMENTARY NOTES

This document has been approved for public release and sale; its distribution is unlimited.

91-19209

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unlimited distribution

13. ABSTRACT *(Maximum 200 words)*

This final report describes progress made by TIS Incorporated for the period 7/1/91 - 10/1/91 on SBIR contract F49620-91-C-0055 toward the development of novel optical computer architectures and supporting methods for exploiting free-space reconfigurable interconnects. Major findings include: (1) Reconfigurable interconnects can reconfigure slower than the bit rate and still improve performance as long as throughput is maintained after reconfiguration; (2) A fixed control sequence does not preclude the use of runtime conditionals, so that the performance of traditional general purpose computing can be improved; (3) A system that uses reconfigurable interconnects is likely to be larger than a functionally equivalent system that does not use reconfigurable interconnects; (4) A reconfigurable approach is most effective for a small active portion of a computer, and is not needed for an entire computer in order to appreciate a performance gain; (5) A reconfigurable interconnect technology can have a significant impact on interconnection networks used in parallel processors; (6) A fixed control sequence must have some level of repetition in order to be practical; and (7) The dataflow model of computing, which theoretically supports maximum parallelism but suffers performance sacrifices in electronic implementations, may be significantly improved since the architecture can be modified to suit the dataflow graph.

14. SUBJECT TERMS

optical computing; computer architecture; reconfigurable interconnects; free-space interconnects; dataflow

15. NUMBER OF PAGES

64

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to *stay within the lines* to meet *optical scanning requirements.*

Block 1. Agency Use Only *(Leave blank).*

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	PR	- Project
G	- Grant	TA	- Task
PE	- Program Element	WU	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known).*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD	-	See DoDD 5230.24 "Distribution Statements on Technical Documents."
DOE	-	See authorities.
NASA	-	See Handbook NHB 2200.2.
NTIS	-	Leave blank.

Block 12b. Distribution Code.

DOD	-	Leave blank.
DOE	-	Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA	-	Leave blank.
NTIS	-	Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only).*

Block 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

TIS Incorporated

Subject: **Novel Optical Computer Architecture Utilizing Reconfigurable Interconnects - Final Report**
SBIR Phase I Contract No. F49620-91-C-0055

Date: **October 1, 1991**

From: **Miles Murdocca**
TIS Incorporated
380 Farmingdale Road
Jackson, NJ 08527
murdocca@cs.rutgers.edu
TR-100191.01

TABLE OF CONTENTS

Abstract 2
1 Introduction 2
1.1 Statement of work, timetable, and schedule of reports 3
1.2 Expenditure of resources 4
1.3 Motivation for exploring reconfigurable interconnects 5
1.4 Background on the optical computing model 6
2 Fixed control sequences 9
2.1 The fixed control sequence model 9
2.2 Fixed control sequences with reprogrammable masks 13
2.3 Fixed control sequences with steerable interconnects 20
2.4 Fixed control sequences with branching object code 20
2.5 Reconfigurable logic 21
2.6 Discussion 25
3 Runtime selected control sequences 26
3.1 The runtime selected control sequence model 26
3.2 An example: the pivoting problem 26
3.3 Hardware caching 28
3.4 Discussion 29
4 Runtime generated control sequences 29
4.1 The runtime generated control sequence model 30
4.2 An example: a runtime generated floating point processor 30
4.3 Discussion 33
5 Applications 34
5.1 Fault tolerance 34
5.2 Parallel processing 36
5.3 Dataflow computing 41
5.4 Scalable Gaussian elimination processor 50
6 Putting it all together 51

TIS - RELEASED FOR DISTRIBUTION

7 Plans for continuing to a Phase II SBIR effort 52
7.1 Software development tools 52
7.2 A hardware prototype 53
8 Collaborations and coordination of efforts 53
8.1 Interactions with Rutgers University and Rome Laboratory 54
8.2 Interaction with Photonics Research Incorporated 54
8.3 Interaction with AT&T Bell Labs 54
8.4 Coordination with existing TIS Phase II SBIR effort 55
9 References 56
Appendix A: Boolean logic 59
Appendix B: Equivalence of AND-OR and OR-NOR logic 62
Appendix C: Gaussian elimination 63

ABSTRACT

This final report describes progress made by TIS Incorporated for the period 7/1/91 - 10/1/91 on SBIR contract F49620-91-C-0055 toward the development of novel optical computer architectures and supporting methods for exploiting free-space reconfigurable interconnects.

Major findings include: (1) Reconfigurable interconnects can reconfigure slower than the bit rate and still improve performance as long as throughput is maintained after reconfiguration; (2) A fixed control sequence does not preclude the use of runtime conditionals, so that the performance of traditional general purpose computing can be improved; (3) A system that uses reconfigurable interconnects is likely to be larger than a functionally equivalent system that does not use reconfigurable interconnects; (4) A reconfigurable approach is most effective for a small active portion of a computer, and is not needed for an entire computer in order to appreciate a performance gain; (5) A reconfigurable interconnect technology can have a significant impact on interconnection networks used in parallel processors; (6) A fixed control sequence must have some level of repetition in order to be practical; and (7) The dataflow model of computing, which theoretically supports maximum parallelism but suffers performance sacrifices in electronic implementations, may be significantly improved since the architecture can be modified to suit the dataflow graph.

1. Introduction

Nearly all digital computers are constructed with electronic technologies that use transistor-based logic gates for switching and use wires to carry information. The arrangements of wires and logic gates define the computer architecture, which remains fixed once a computer is created. Programs are then constructed in such a way that they map onto the fixed computer architecture, and the responsibility for constructing programs in this way is typically handled by a compiler that translates a high level language such as C or Pascal into object code for the instruction set of the target machine. Often, this mapping is not very good since the physical architecture cannot be modified to suit particular computer programs. Although some fine-grained parallel computer

architectures such as the Connection Machine [1] allow for programmability at very low levels of complexity, the architecture itself remains fixed and performance is sacrificed when compared to specialized hardware designed to carry out specific tasks. The Phase I project explores an optical computing model that supports gate-level reconfiguration of the interconnects, which offers the novel capability of changing the architecture during the course of computation. Problems addressed in this report include the development of three models that cover different aspects of reconfigurable interconnects, the identification of potential applications of the models, and a plan for continuing to a Phase II effort.

1.1 Statement of work, timetable, and schedule of reports

Three reconfigurable interconnect models are explored in the Phase I effort. The first model makes use of fixed control sequences, in which the interconnects are changed in a predetermined sequence that remains fixed throughout a computation. For this model, there is no need to feed the results of a computation back to the reconfiguration mechanism. Two variations of this model are considered, one in which a fixed interconnect structure is used and a reconfigurable mask modifies the interconnect, and one in which beam-steering elements are used instead of masks. The second model selects a control sequence based on the results of previous computations. For this model, a fixed number of precompiled control sequences are provided, and the results of computations determine which sequence is applied next. For the third model, the results of computations are fed into a mechanism that generates new control sequences based on the needs of the running program.

The fixed control sequence study focuses on applications with fixed control streams such as in the use of Gaussian elimination in solving systems of linear equations, which is important for null steering in phased array radar applications [2, 3]. This aspect of the Phase I effort is an extension of a Phase I Rome Laboratory (RL) SBIR contract (F30602-90-C-0081, project engineer is Robert Kaminski, 315-330-4092) performed by TIS which involved the design of an architecture for a digital optical Gaussian elimination processor. A Phase II follow on effort (F30602-91-C-0101, project engineer is Robert Kaminski) is in progress and will be coordinated with a proposed Phase II follow on to this Phase I effort, as described in Section 8. The existing Phase II effort develops a fixed interconnection technology for the processor. The runtime-selected study focuses on applications with precomputed control sequences that vary in data-dependent ways, such as in using addition for one time interval and then using subtraction for another time interval. For this study, the results of conditionals from running programs determine which precomputed control sequences are used and in what order. The final study explores program-generated sequences, in which the form of the control sequence is not known until execution time, which is when the sequence is generated. An application of this paradigm is to dataflow machines, which is discussed in Section 5.3.

A significant aspect of the reconfigurable interconnection schemes explored here is that computer architectures are modified on-the-fly after a system is placed in service. This allows for greater flexibility in fault tolerance, hardware re-use, and in generating custom architectures to solve

A-1

specific problems without actually touching the physical hardware.

For this final report, progress for the contract period 7/1/91 - 10/1/91 is detailed in accordance with the schedule shown in Figure 1.

Task or deliverable

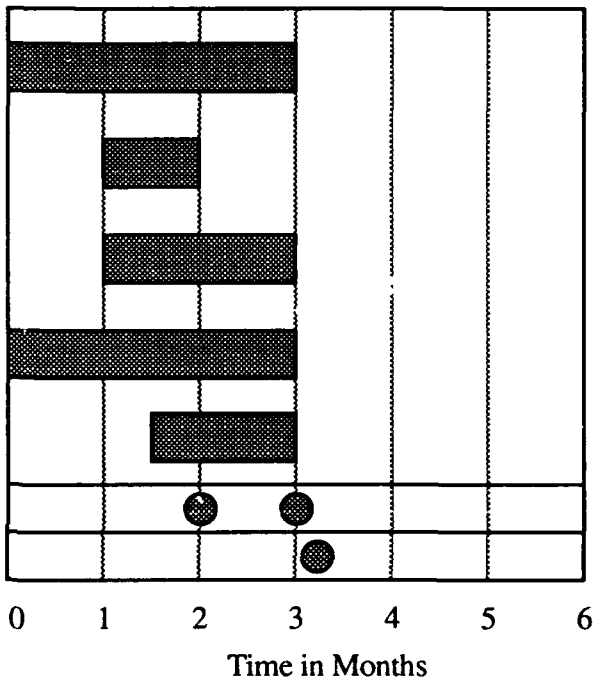


Figure 1: *Timetable for the Phase I effort.*

The schedule of reports for the Phase I SBIR effort is listed below. Note that the Final report is dated three months earlier than the scheduled report date of December 31, and that the interim reports are also dated earlier than the scheduled dates.

Report	*Scheduled*	*Actual*
Interim Report #01	August 31, 1991	August 17, 1991
Interim Report #02	October 31, 1991	September 30, 1991
Final Report	December 31, 1991	October 1, 1991 <--- This report

1.2 Expenditure of resources

All of the labor resources have been applied since the start of the contract. Principal investigator (PI) Murdocca contributed 360 hours to the effort, and the results of Murdocca's effort are detailed

here. Professors Saul Levy and Apostolos Gerasoulis of the Computer Science Department at Rutgers University contributed 60 hours (Levy) and 80 hours (Gerasoulis) for studies into dataflow computing (Levy) and the more general problem of interconnects in parallel processing (Gerasoulis). Edward Roos (an optomechanical designer under contract to TIS, formerly with OptiComp Corporation) contributed 60 hours to the investigation of acoustooptic modulators for reconfiguration.

Murdocca visited RL as a TIS employee during July 9-10, 1991, for a technical exchange with members of the Photonics Center. Discussions covered anticipated milestones for the S-SEED processor, and the involvement of microlaser expert Jack Jewell in the effort. Roos visited RL with Murdocca and Professor Thomas Stone (Rutgers U.) on September 18, 1991 to coordinate a potential follow on to this effort with the RL effort.

1.3 Motivation for exploring reconfigurable interconnects

A digital circuit must account for all possible input combinations that may arise during the course of computation even though only one input combination exists at a time, so that much of the logic is underutilized. If some information is known about a computation regarding the complexity of logic that is needed on each time cycle, then greater efficiency can be realized through a mechanism that reconfigures the circuit during operation (see Section 2.4).

For conventional electronic digital circuits, the gate-level interconnection network is fixed when the system is created. This implies that the configuration of the hardware is always present even when large parts of it are idle. A good numerical computer provides floating point operations as well as integer operations. Floating point is often enhanced with hardware transcendental functions. When the computer is performing integer operations, it will not use the floating point hardware. When floating point multiplication is performed, transcendental hardware sits idle. With fixed interconnects logic is underutilized. If the wires in electronic circuits can be changed on demand, then small circuits can be made to yield the same performance as large circuits. There are probably no reasonable means for doing this in electronics, but if the interconnects in a free-space digital optical computer can be changed on demand, then fewer gates can perform the functions of many, so that an optical computer can potentially provide an architectural advantage over electronic computers.

Reconfigurable interconnects hold promise for improving the performance of parallel processors as well. A switching network is commonly used for interconnecting large numbers of processors. The switching network increases communication delays which can have a profoundly negative effect on performance. A reconfiguration strategy can achieve the same goal without introducing a large latency. This application of reconfigurable interconnects is discussed in further detail in Section 5.2.

Although these arguments suggest that a greater efficiency can be achieved with a reconfigurable

interconnect technology, there are special cases where little or no improvement can be made. For decisionless computation, like matrix multiplication, conventional hardware implementations can be pipelined at the gate level if engineered properly. Little savings can thus be achieved in reconfiguring the gate-level interconnects for these applications since the logic sequence is essentially laid out in space and is utilized nearly 100% of the time due to the nature of the application. However, when there is a need to modify the sequence, for example, to increase precision or to isolate faults, then there may be an advantage. Thus, the motivations for exploring a reconfigurable interconnect technology include performance improvements, as well as fault tolerance and post-fabrication modifications.

1.4 Background on the optical computing model

The model of a digital optical computer that is used here is based on the all-optical S-SEED [4] processor developed at AT&T Bell Labs [5] although different devices than the S-SEEDs may be considered for a Phase II hardware prototype (see Section 8.2). Figure 2 illustrates a digital optical computing model that is similar to the AT&T configuration. The model is composed of alternating arrays of optical logic gates and free-space regular interconnects. Masks in the image planes block light at selected locations which customize the interconnects to perform specific logic functions such as addition or sorting. The system is fed back onto itself and an input channel and an output channel are provided. Feedback is imaged with a vertical shift so that data spirals through the system, allowing a different section of each mask to be used on each pass. Optical hardware for implementing the logic and interconnects is described below.

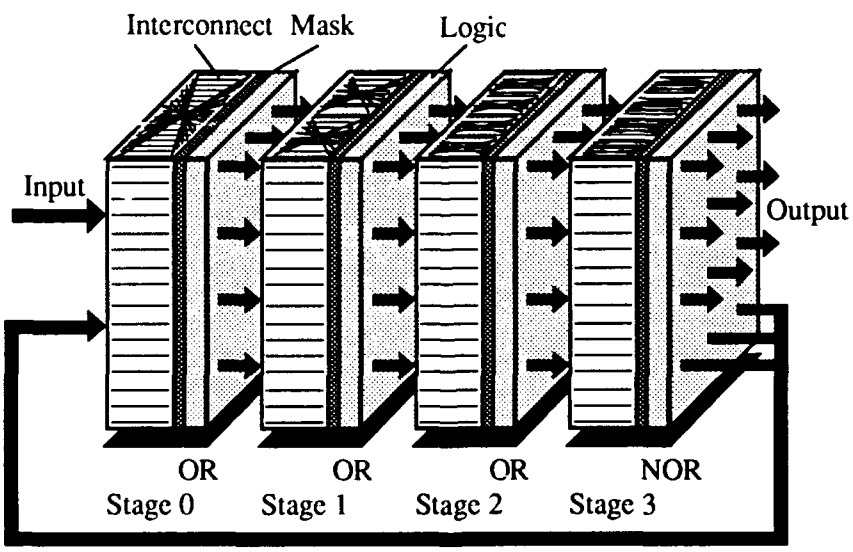


Figure 2: *Arrays of optical logic gates are interconnected with optical crossovers [6]. Fixed masks in the image planes block light at selected locations which customize the system for specific logic functions such as addition or sorting.*

1.4.1 Gate-level optical interconnects: Figure 3 shows a schematic diagram of the optical crossover interconnect [6], which is one of a number of demonstrated methods of interconnection that is suitable for this model. An array of input beams is split into two identical copies. One copy is imaged onto a mirror and is reflected back through the system to the output plane, while the other copy is permuted according to the period of the prism array. The combined copies are displaced slightly with respect to each other so that each copy can be independently masked in the output plane. The gate-level interconnection pattern that this interconnect achieves is shown for varying periods of the prism arrays in the interconnection stages of Figure 2.

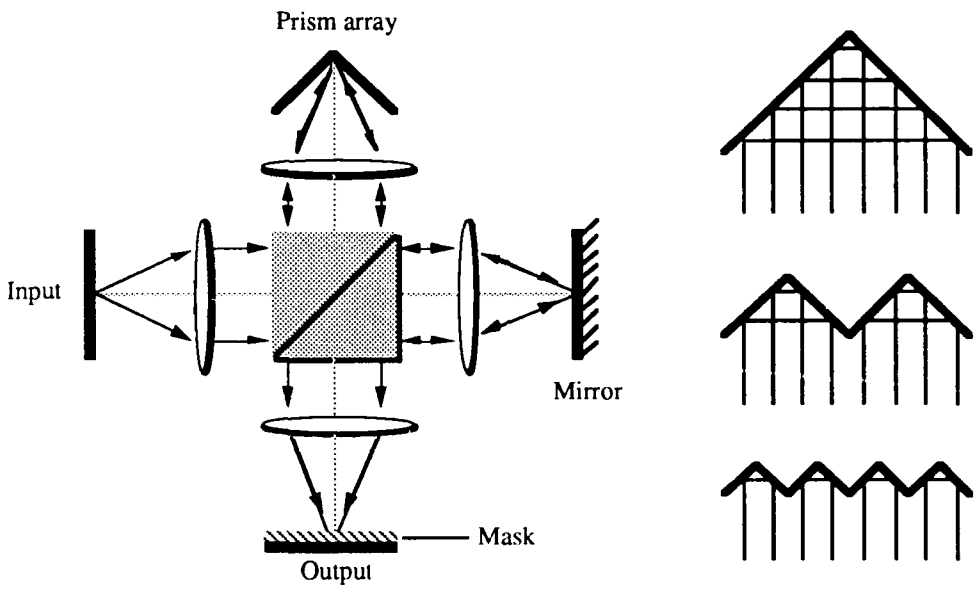


Figure 3: *Optical crossover interconnect. A two-dimensional array of input beams is split into two identical copies. One copy is imaged onto a mirror and is reflected back through the system to the output, while the other copy is imaged onto a prism array that permutes the beams according to its period. Connection paths achieved with different prism array periods are shown in the right panel.*

This particular implementation of the crossover supports a fixed set of connections that are customized through fixed masks in the image planes. If the fixed masks are replaced with reprogrammable masks then this approach will support reconfiguration. Another method of interconnection that supports reconfiguration is beam-steering, provided for example with acoustooptic modulators [7] or with photorefractives [8]. Beam-steering can be more light efficient in principle since beams are steered to their targets instead of being selectively blocked, but reconfiguration time is generally slower than the bit rate.

1.4.2 Optical logic gates: A number of optical logic gates can be used to support the reconfiguration model, and a few promising devices are described here. The symmetric self-electrooptic effect device (S-SEED) [4] is a more recent version of the SEED which is used in optical processor testbeds at AT&T Bell Laboratories in Holmdel, New Jersey and at Bell Labs in Naperville, Illinois, and in a similar testbed under development in the Photonics Center at RL, which involves collaborations between Rome Laboratory, Rutgers University, and TIS. The SEED is based on an electrically coupled optical modulator and detector pair. The device is made up of approximately 1200 alternating layers of GaAs and GaAlAs in an 8μm thick quantum well structure placed inside a PIN photodiode detector as shown in Figure 4. When light is applied to the detector, a current is generated that reduces the potential across the quantum well. When a strong enough current is created, the positive feedback allows the device to retain its state after the light source is removed. One of the operating modes of the device is to pass light of low intensity and to absorb light of high intensity, implementing negating logic. The electrical properties of the device make it easy to use in the laboratory, and since communication is handled optically, the system speed of a computer made up of these devices is limited only by the device speed. Expected operating rates are several hundred megahertz, although current devices operate only in the tens of megahertz range due to the lack of sufficient optical power at 850nm from a single laser source. A fabricated array of S-SEEDs is shown in Figure 5.

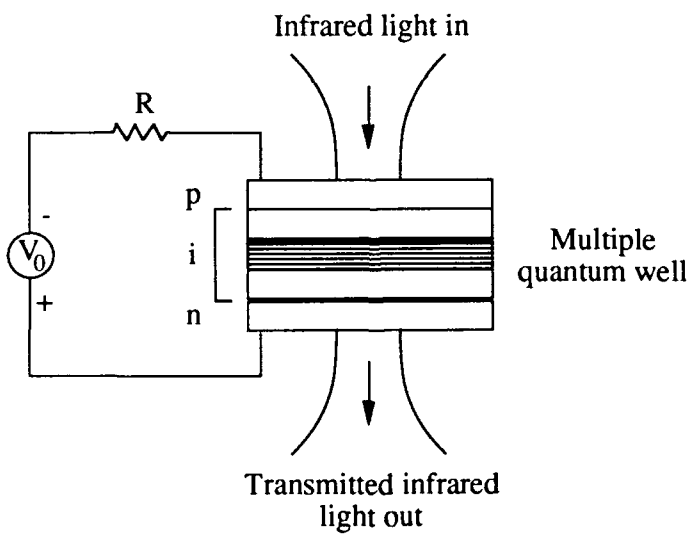


Figure 4: *Schematic of the self-electrooptic effect bistable device.*

Another optical logic gate that is based on multiple quantum well (MQW) technology is the surfaCe Emitting Laser Logic device (CELL) [9] which is based on Heterojunction PhotoTransistors (HPTs) and Vertical-Cavity Surface-Emitting Lasers (VCSELs) [10] as shown in Figure 6. An early array of VCSELs is shown in Figure 7, and a similar VCSEL structure is used for the CELL devices. The CELL operates by allowing a low intensity signal on the HPT to create a

photocurrent which is electronically amplified to a level large enough to drive the microlaser above threshold. Thus an optical-in/optical-out device is created which combines desirable attributes of both transistors and lasers.

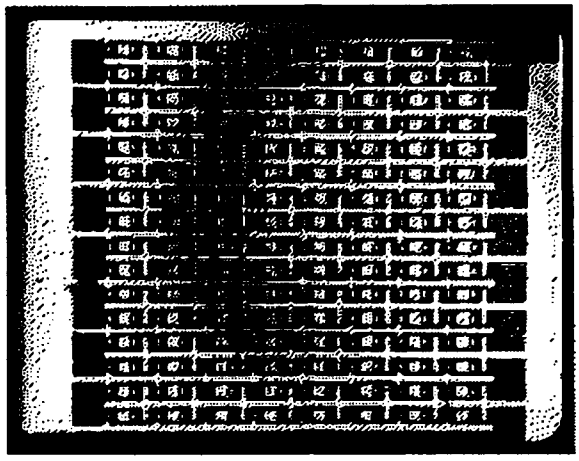


Figure 5: Array of S-SEEDs with a 40μm spacing between mesas.

VCSELs have been fabricated and demonstrated in two-dimensional arrays, and CELLs are in the process of being fabricated at Sandia National Laboratories. Dr. Gregg Olbright was the principal investigator involved in this work at Sandia before he and Dr. Jack Jewell, formerly of Bell Labs, formed Photonics Research Incorporated (PRI) where they continue in this line of work. Olbright and Jewell maintain frequent contact with Technical Imaging Services on the use of these devices for the Phase I effort (see Section 8.2).

2. Fixed control sequences

A number of computer applications perform fixed sequences of operations that do not depend on the data being processed. A simple traffic light controller repeats the fixed sequence: green-amber-red-green (the convention used in the United States) with fixed time intervals, and since this sequence continues regardless of traffic conditions, it is data-independent. A traffic light that alters its behavior based on traffic volume or the presence of walk-light requests is a data dependent application since its behavior is not known until traffic volume changes and walk-light requests are generated. This section is devoted to data-independent, fixed control sequences. The motivation for considering this special type of computation is that it simplifies the control sequence mechanism. The control sequence is generated external to the system and therefore does not introduce delay in the processing loop as is the case for the models discussed in Sections 3 and 4.

2.1 The fixed control sequence model

The general form of the fixed control sequence model consists of a logic stage followed by an externally controlled interconnect stage as shown in Figure 8. A data input stream and an

interconnect control stream are created externally and are imaged into the system. There is no feedback from the output to the control stream mechanism as there is for the remaining two models since the sequence is known *a priori*, although there is a feedback path from the output to the data input stream so that iterative computing is supported. For the fixed control sequence model, two variations are considered: one in which the interconnects are controlled with reprogrammable masks (Section 2.2) and one in which the interconnects are controlled with beam-steering elements (Section 2.3).

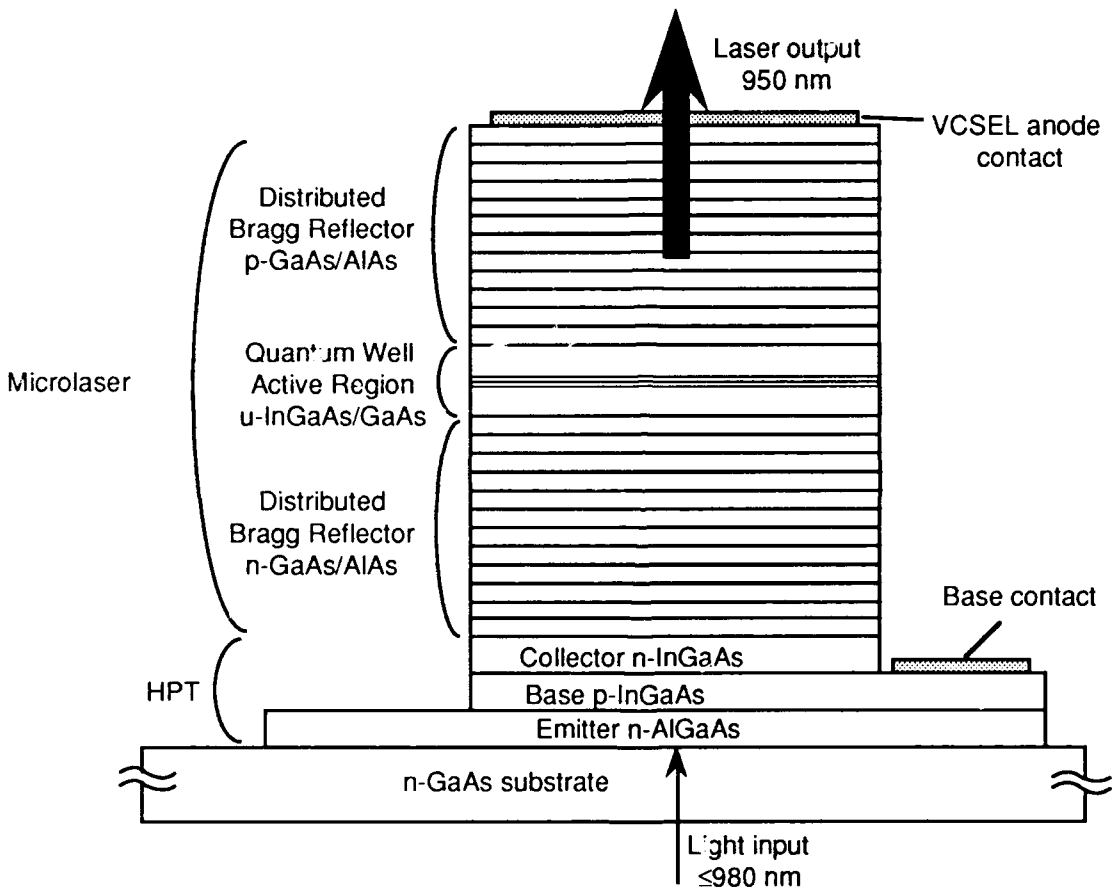


Figure 6: *Structure of a CELL having an applied base contact. The emitter contact is through the substrate at a potential common to the other CELL emitters in the array.*

As an example of how this model influences computation, consider the logic circuit[1] shown in

[1]The layout was generated automatically from design tools created at Rutgers University under joint AFOSR/ONR support.

Figure 9, which implements the majority function (see Appendix A). The stage indices on the left correspond to the logic and interconnect stages of the optical computing model shown in Figure 2. There are four stages, and two passes are made through the four stages, so the circuit is eight levels deep. Inputs are at the top and the dual-rail output is produced at the bottom. The boxes represent optical logic gates, which is a variation of the representation normally used in digital circuit designs. Normally the logic gate representation described in Appendix A is used, but the box notation is used here to simplify diagrams, and because there is a need to distinguish between the signals that an optical logic gate drives (left and right), unlike an electronic approach where the signals are common.

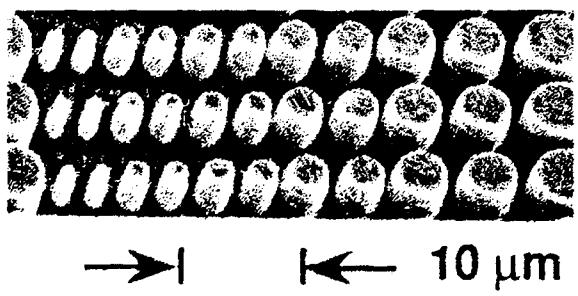


Figure 7: Scanning electron micrograph of a small portion of an array of vertical-cavity surface-emitting lasers [10].

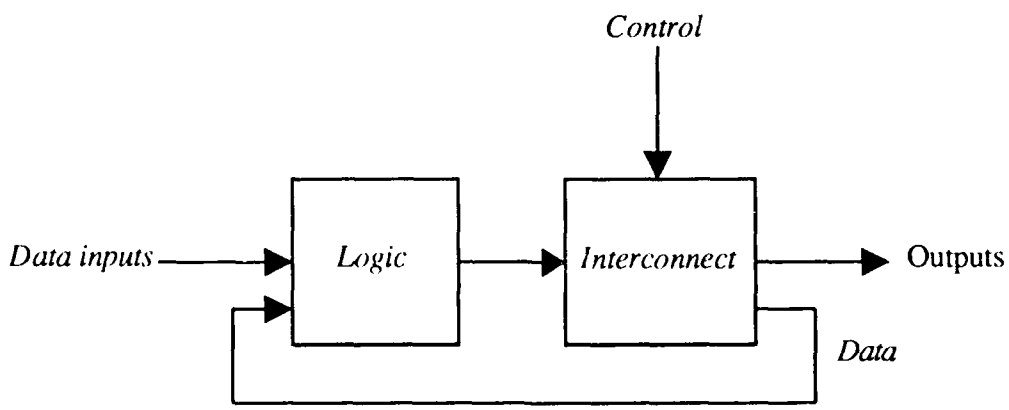


Figure 8: *Block diagram of the fixed control sequence model. Data inputs and control inputs are imaged into the system from external sources in a sequence that is fixed when computation begins.*

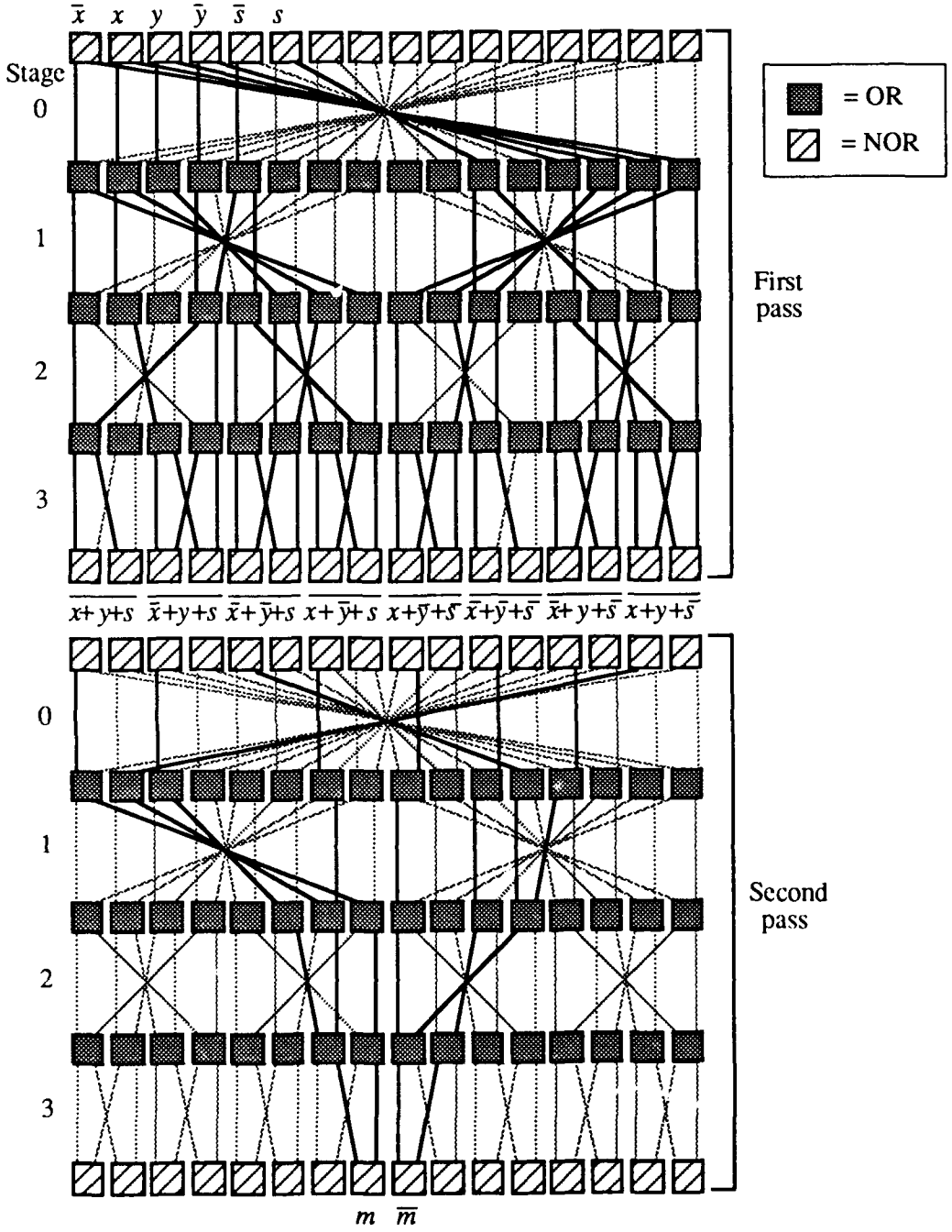


Figure 9: *The majority function is implemented on an optical programmable logic array (PLA).*

Now consider the two systems shown schematically in Figure 10. The system on the left represents the 16 wide by 8 deep circuit shown in Figure 9. The system on the right represents a

circuit that performs the same function in just one level by recirculating data and modifying the control mask. For the system on the right, less throughput[2] is supported because the circuit has to perform the tasks of the original eight circuit levels. Thus, there is a direct tradeoff between the amount of physical logic that is used and the degree of throughput that is supported. However, when the physical logic is reduced, then a smaller system results, and small systems are preferred to large systems since lens fields are smaller, distortions are smaller, and other aspects of the physical implementation are simplified. These advantages are summarized in the work of Jewell *et. al.* [11] as shown in Figure 11. A conclusion that may be extrapolated is that it is better to use many copies of a small system when throughput is an issue than it is to work with a single functionally equivalent large system, and further, a system that supports a reconfigurable interconnect technology has a greater opportunity to exploit this property than a system that does not.

2.2 Fixed control sequences with reprogrammable masks

A configuration for fixed control sequences is considered in which the interconnects are fixed but mask patterns that customize the interconnects are allowed to change. As an example of this approach, consider the truth table shown in Figure 12 which defines six Boolean functions in three variables. The sum and carry functions describe the functional behavior of a full adder, which is a basic building block used in creating an addition unit. The sub and borrow functions describe the functional behavior of a full subtracter. The majority function (see Appendix A) is true (logical 1) whenever more than half of its inputs are set to 1. The parity function is true whenever there is an even number of 1's in the input (for even parity). The majority and parity functions are used in error correction (see Section 5.1).

Assume that these functions need to be implemented on some unspecified processor. As in most conventional electronic computers, the functions are implemented in a fixed structure that is always present even though only one of the functions (or a pair, such as sum-carry or sub-borrow) is used at any one time. A potential improvement to the conventional electronic approach is to implement just one or two functions at a time, and to reconfigure the circuit to implement the one or two functions that are needed on demand. Although the investigation reported here considers the development of an optical reconfiguration technology, it may be possible to use an electronic approach rather than to resort to optical technology in order to achieve this behavior. For example, consider the general form for a conventional electronic three variable, two function programmable logic array (PLA) shown in Figure 13. Three Boolean variables a, b, and c enter at the top, and

[2]Throughput refers to the amount of information that is passed through a system. A system that has no feedback path supports a throughput rate equal to the throughput of the slowest stage. A system with feedback supports a reduced throughput rate, which is equal to the throughput of the slowest stage divided by the number of times the feedback path is taken.

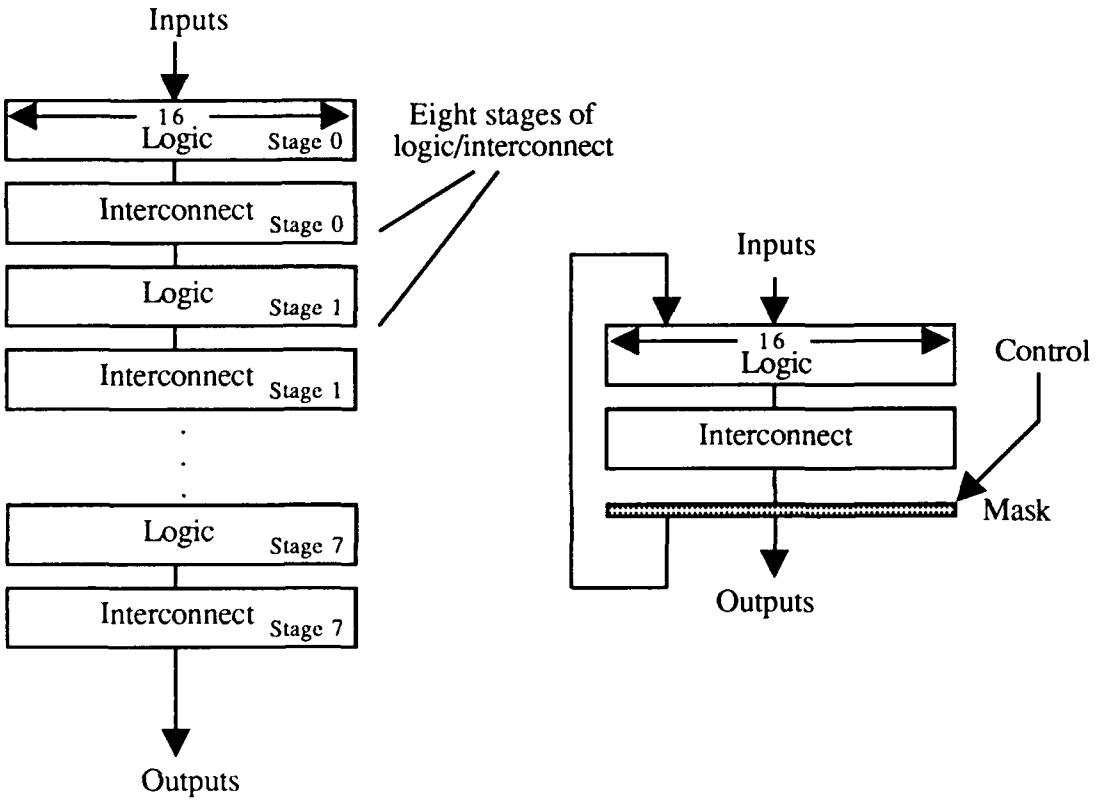


Figure 10: *Eight-stage logic/interconnect system (left) and equivalent folded system using a fixed control sequence (right).*

Devices per array	$N^2 \rightarrow N^2/s^2$
Number of arrays	$M \rightarrow s^2M$
Power per array	$P \rightarrow P/s^2$
Intensity	$I \rightarrow I$
Lens diameter	$D \rightarrow D/s$
Lens aberrations	$A \rightarrow A/s$
Latency	$\tau \rightarrow < \tau/s$
Temperature rise	$\Delta T \rightarrow \Delta T/s$

Figure 11: *The effects of scaling a system down by a scaling factor* s *to microoptic sizes while maintaining equivalent computational power are summarized (adapted from Jewell,* et. al., *[11]).*

these signals as well as their complements (produced by the three inverters) are passed through to eight AND gates. The crosspoints occur where data inputs intersect AND gate inputs and where AND gate outputs intersect OR gate inputs, and have one-time programmable fuses that enable or disable connections between the intersecting lines. The AND matrix is followed by a programmable OR matrix. Note that each AND gate has six input lines, that the single input line shown represents six, and that a similar simplification is used for the OR matrix. The AND-OR matrix provides all of the computational power needed to realize any two functions f and g of three Boolean variables a, b, and c.

a	b	c	sum	carry	sub	borrow	majority	parity
0	0	0	0	0	0	0	0	1
0	0	1	1	0	1	1	0	0
0	1	0	1	0	1	1	0	0
0	1	1	0	1	0	1	1	1
1	0	0	1	0	1	0	0	0
1	0	1	0	1	0	0	1	1
1	1	0	0	1	0	0	1	1
1	1	1	1	1	1	1	1	0

Figure 12: *A truth table describes six functions in three variables.*

It is a relatively simple task to replace the one-time programmable fuses with reprogrammable links, so that all six of the functions described in Figure 12 can be implemented with a one or two-function reconfigurable PLA in an electronic technology. Thus in principle there is little motivation for considering the use of a less developed optical technology. However, it is difficult to set the crosspoints quickly in an electronic implementation due to bandwidth limitations of communicating information to and from an integrated circuit [12]. The same crosspoint information can be stored off the chip and can be brought in through the pins, but the simple PLA shown in Figure 13 would need at least 64 crosspoint pins (6x8 = 48 pins for the AND stage and 2x8 = 16 pins for the OR stage) plus the power pins and the control pins, for what is a very small circuit. Pin counts for very large chips go only as high as about 256. A conventional VLSI chip might contain hundreds of such PLAs, and even though they may not all need to communicate to and from the chip, a nearly hopeless pinout problem is still posed for circuits of reasonable complexity. A possible solution is to place a small memory at each crosspoint that stores the control sequence so that there is no need to bring in the sequence from an external source. Although this approach is possible, it forces the diameter of the circuit to increase, which consumes chip area and increases delay since some functions will inevitably be pushed off of the chip. An optoelectronic approach may offer a solution rather than using an all-optical approach, but then the electronic portion is fixed, although this may not be a problem for some applications as reported in Ref. [13].

2.2.1 Size of the stored sequence For the proposed reconfigurable interconnect model, the potential size of the external mechanism should be considered. For the PLA example described above, a small predetermined control sequence is repeated. 48 AND stage crosspoints and 16 OR stage crosspoints need to be enabled and disabled on each clock cycle, for a total of 64 bits of information imaged into the PLA per cycle. For the two competing systems represented in Figure 10, an eight-to-one reduction in system size is suggested as a result of using reprogrammable masks. The PLA case considered here maps onto the eight stage structure, in which four 16-wide stages make up one pass through the circuit shown in Figure 9, so that a total of (16x4 = 64) bits x 2 = 128 bits of information must be stored external to the system for each pass through the PLA, possibly in a storage loop or in some recording material that has a fast read time. Although 128 bits per PLA cycle adds up to a large amount of external storage after a few hundred cycles, many if not most applications exhibit a significant amount of repetition so that only a few unique sequences need to be stored.

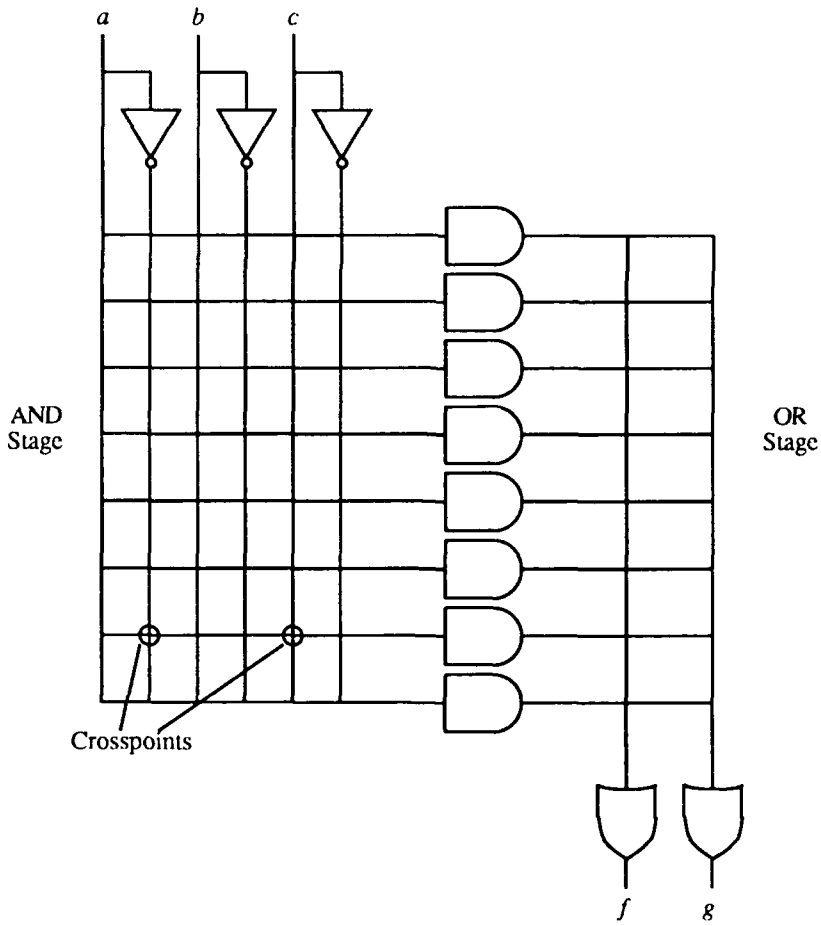


Figure 13: *A three-input, two-output PLA has 48 crosspoints in the AND stage and 16 crosspoints in the OR stage.*

For the case, however, in which the control sequence is not repetitive, then an extremely large storage mechanism is needed, or a device is needed that generates control masks at the bit rate of the switching devices. Storage for a nonrepetitive computation appears to be a fruitless endeavor since the space requirement approaches infinity quickly. A control mask generation mechanism seems more appropriate as long as it keeps pace with the devices. This, however, also appears to be a fruitless endeavor since a number of logic operations will be needed to generate a single mask pattern in ali but the most trivial cases. The logic gates being controlled should use the fastest available switching technology (otherwise, there may be little performance improvement in using optical processing), so that it is unrealistic to expect a sequence generation mechanism to use even faster logic than the devices being controlled. However, note that the case in which a control sequence contains no repeat pattern is very rare, and that the sequence probably cannot run for very long regardless of reconfigurability. The reason for this is that a compiled instruction sequence is normally used to control a computation, and the length of the instruction sequence mirrors the execution time when the sequence has no repetition. For example, if a computer that is capable of executing 50 million instructions per second (MIPS) runs for 10 seconds, then storage must be provided for a 50MIPS x 10s = 500 million word instruction sequence. This situation is so abnormal that it is not considered here. No suggestion is made that such a sequence even exists that is useful, although it is probably not impossible that such a sequence does in fact exist. Summarizing, a control sequence must have some degree of repetition in order to exploit the fixed control sequence model. This restriction is simply a property that already exists in conventional computing.

Continuing with a study into the size of the mechanism that generates control sequences, notice that although the mask patterns may be very large, that a small indexing mechanism can be used to invoke the patterns so that a high level compiler can simply generate indices to these patterns rather than require storage of many identical copies of the masks, similar to the way that pointers are used for passing arrays as arguments to subroutines in computer languages such as C or FORTRAN, or to the nanoprogramming concept used in some microcontrolled processors [14]. In order to take advantage of this scenario, a mechanism is needed to translate an index into a mask pattern, which again invokes the problem of needing faster control logic than the devices being controlled. However, since the control sequence is independent of the data being processed, a mask generation pipeline can be created and maintained without generating wait states that might otherwise render the approach ineffective, so that it is only necessary for the controlling logic to be as fast as the logic being controlled. Thus, it appears reasonable to generate traditional object code and to use the object code to select mask patterns.

2.2.2 Savings in active computing area Given that interconnection control information can be imaged into the active computing area[3] with reasonable complexity, a question that needs to be addressed is how much active chip area is saved with a reconfigurable approach. It appears unlikely that total area will be reduced, since external storage must be provided for the crosspoint settings that are moved from the active chip area, and because a mechanism must be provided that images externally stored crosspoint settings into the active computing area. However, it is important that the active computing area be reduced, even if the total area increases, since the speed of this region places an upper bound on the speed of the entire processor.[4] Consider again the six function truth table shown in Figure 12 and the two-function PLA shown in Figure 13. A traditional electronic PLA implementation of the six functions can be made without using a reconfiguration technology as shown schematically in Figure 14. Crosspoints marked with heavy dots indicate positions of enabled connections. Notice that the number of crosspoint settings in the OR stage has grown from 2x8 = 16 for the two-function PLA shown in Figure 13 to 6x8 = 48 crosspoints. In addition, the outputs of the AND gates must now be capable of driving up to six loads instead of only two. Again, all six functions are unlikely to be used simultaneously in a computer, so that the added expense of increasing the active computing area to accommodate all six functions does not increase the performance of the individual operations. For the reconfigurable approach, the same functionality is provided at the smaller price of implementing only two of the functions, thus resulting in a performance improvement since the active logic is placed in a smaller area, with smaller fan-in and fan-out requirements. A reduction of 67% is thus suggested for the OR stage of the example PLA.

A consideration in supporting this claim is in understanding how the needs of various functions are distributed. For example, if there is a long run of addition operations followed by a long run of subtraction operations, then a mechanism that reconfigures slowly with respect to the bit rate may be effective, in the same way that the slow process of loading a cache from main memory is offset by the overall performance improvement attributed to the locality principle. If however, the

---

[3]Nearly all of the time, there is some portion of unused logic. The active computing area refers to the set of logic gates that are involved in the current computation.

[4]An analogy can be drawn to cache memory, which is held local to the processor, while the bulk of main memory is placed in an external area of a computer, typically on the system bus. The processor still operates as if all of main memory is fast, local cache, because computation is typically localized in nature. That is, it is unusual for memory accesses to be randomly distributed over the memory space. Moreover, even if the expense is allowed for making all of main memory as fast as the cache, there would be little improvement in performance because of the locality principle, and because even with fast logic the main memory will have to operate more slowly because of its sheer size. A cache is typically on the order of 1/256 of the size of main memory [15].

distribution of functions is characterized by frequent changes, then a reconfiguration mechanism must respond at the bit rate in order to be effective. In actuality, both forms of computation occur. A summary of the approaches and a recommendation for a potential configuration for a proposed Phase II effort are discussed in Sections 6 and 7.

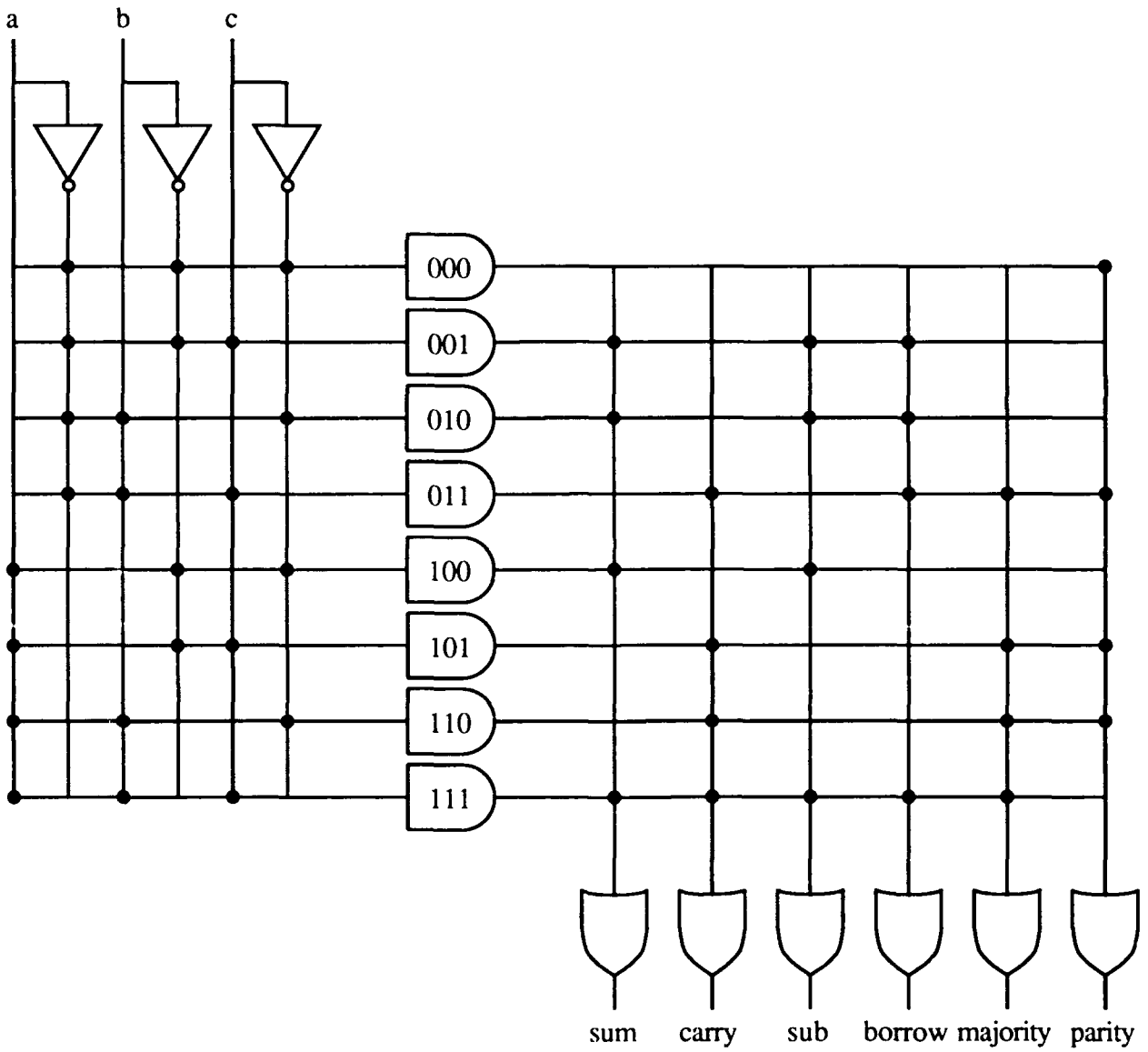


Figure 14: *A conventional layout of a three-variable, six function PLA is shown. Enabled connections are marked with heavy dots. The functions correspond to the truth table shown in Figure 12.*

2.3 Fixed control sequences with steerable interconnects

One problem with reconfiguring the interconnects by modifying the mask patterns is that light is wasted since all anticipated connections are hardwired into the structure, and then unwanted signals are selectively blocked. Thus, even though fan-in and fan-out constraints are improved for the OR matrix when masks are reconfigured, the AND matrix is still burdened with generating enough fan-out for all minterms. This problem can be reduced if the physical beams of light are selectively redirected to their targets, since there is no longer a need for each input signal to power all of the AND gates and there is no longer a need to rely on masks to enable and disable connections. Potential advantages of this approach include not only improved fan-in and fan-out, but also that fewer levels of logic may be necessary to implement functions since circuits do not have to be mapped onto a fixed set of interconnects.

Two variations of this approach are considered: (1) interconnects that change at the bit rate, and (2) interconnects that change more slowly. The computing situations that are appropriate for each approach appear to be identical to the situations for reprogrammable masks, namely that slow reconfiguration is effective for long runs of identical computations, and that fast reconfiguration is needed for applications with rapidly varying needs. A significant difference is that although there are a number of possible implementation strategies for interconnects that reconfigure slowly such as the use of acoustooptic modulators [7] or photorefractives [8, 16], there appears to be no potentially viable technology that supports beam-steering at the bit rate. Although this situation creates a technology gap, beam-steering in general need not compete with the reprogrammable mask approach, in fact, the reprogrammable mask approach may be appropriate for parts of a computation that change computing needs quickly, while the steerable interconnect approach may be appropriate for parts of a computation that change more slowly. The reconfigurable interconnect technology is preferred to the reprogrammable mask technology because light is simply steered to its target rather than discarding it through masks, thus reducing fan-in and fan-out requirements of the switching devices, although the interconnection elements may be more complex. An application of the photorefractive approach is explored in Ref. [17] for reducing the number of stages in a switching network. Irregular interconnects are suggested for implementing expander graphs optically. The result is an interconnection network with a shallower depth than can be achieved with more conventional fixed regular interconnects.

2.4 Fixed control sequences with branching object code

The use of a fixed control sequence does not imply that there are no branches in the executing program. This is an important point because the vast majority of computing applications make use of conditional branches. For example, a conditional branch in a computer program may take the form:

```
if (x > y) then y = y + 10;
     else y = y - 10;
```

in which case the flow of control cannot be predetermined since the values of `x` and `y` are not known until execution time. However, on a conventional electronic computer, the condition test `(x > y)` and the assignments `y = y + 10` and `y = y - 10` are implemented in fixed hardware, such as in an arithmetic logic unit (ALU). The ALU remains unchanged for both the addition and the subtraction cases, and only the data flowing through it changes. The function to be executed is determined by a control input. An ALU consists of a number of levels of logic, and these levels do not change throughout the execution of a computer program. It is straightforward to design an ALU in such a manner that only one of its levels are used at any one time without introducing any major sacrifices in performance. It may thus be possible to exploit a reconfiguration technology that implements one level of logic of the ALU at a time, which reduces the active computing area of the ALU. Although arguments supporting reconfiguration have been made previously [17] the effort reported here goes further in showing the improvement in circuit complexity and in circuit latency as discussed below.

An observation that is sometimes made is that most of a computer is idle most of the time. Although this is true in general for von Neumann style architectures (the von Neumann style is found in a conventional general purpose computer like an IBM PC), it is due almost entirely to the large transistor count devoted to random access memory (RAM). RAM is very dense, and consumes a small fraction of computer volume even though it accounts for the bulk of the active components. It is typical that when a computer adds, it does not subtract, nor does it perform a square root at the same time. Thus the argument might be made that most of the central processing unit (CPU) and not just the memory is idle for most of the time. This in fact is *not* true in general. Consider the truth table and logic diagram for the 16-function 74181 arithmetic logic unit (ALU) [18] shown in Figures 15 and 16. Although only one function is selected at a time, there is a great deal of sharing of logic among the 16 functions. Nearly every logic gate is used for the ADD operation which is only one of the functions. Thus, most of a CPU is not wasted as an observer might initially conclude from the frequency of usage of various instructions. However, although most of the logic gates in an ALU may be used most of the time, most of their functionality is wasted most of the time. That is, a four-input AND gate may need only two of its inputs at any time, so that although the gate is still being used, it introduces a cost factor of two in circuit area since it is underutilized. Consider the shaded portion of Figure 17, which indicates the amount of utilized logic actually used for the ADD operation. There are 63 logic gates, of which only 15.78 are used for the ADD operation as indicated in the figure, so 15.78/63 = 25% of the 63 logic gates are needed. The first two of the six gate levels are not needed since they are devoted to function decoding which is eliminated here. This reduces circuit latency by 1/3 in addition to reducing gate count.

2.5 Reconfigurable logic

One problem with reconfiguring the interconnects while leaving the logic gates unchanged is that some efficiency may be sacrificed. For example, the expression AB + AC can be implemented with three logic gates in an AND-OR circuit as shown in the left side of Figure 18. An OR-AND

circuit can implement a functionally equivalent circuit with only two logic gates as shown in the right side of Figure 18. A reconfigurable interconnect approach cannot exploit this property unless the interconnects can selectively skip some of the stages, which is not characteristic of the methods described in this report. However, a technology that allows the logic gates to change form, such as from AND to OR or from OR to AND, offers a potential implementation. A candidate device for this approach is the microlaser based CELL discussed in Section 1.4. According to Dr. Jack Jewell of PRI, which offers the CELL as a product, AND and OR logic can be dynamically interchanged by altering the electrically controlled biases. For conventional electronic computing, logic is often constrained to be of only one type such as NAND or NOR with small fan-ins and fan-outs, and these constraints do not pose a severe burden on performance, since rows of logic can be selectively skipped simply by applying an appropriate wiring pattern. However, since every optical signal is forced to go through every logic array for the optical computing models considered here, it may prove valuable to allow some flexibility in the choice of logic gates.

Selection				Arithmetic operations
S3	S2	S1	S0	M=0, Cn = carry-in
0	0	0	0	F = A PLUS 1
0	0	0	1	F = (A + B) PLUS 1
0	0	1	0	F = (A + B') PLUS 1
0	0	1	1	F = ZERO
0	1	0	0	F = A PLUS AB' PLUS 1
0	1	0	1	F = (A + B) PLUS AB' PLUS 1
0	1	1	0	F = A MINUS B
0	1	1	1	F = AB'
1	0	0	0	F = A PLUS AB PLUS 1
1	0	0	1	F = A PLUS B PLUS 1
1	0	1	0	F = (A + B') PLUS AB PLUS 1
1	0	1	1	F = AB
1	1	0	0	F = A PLUS A PLUS 1
1	1	0	1	F = (A + B) PLUS A PLUS 1
1	1	1	0	F = (A + B') PLUS A PLUS 1
1	1	1	1	F = A

Figure 15: *Truth table for the 74181 4-bit ALU.*

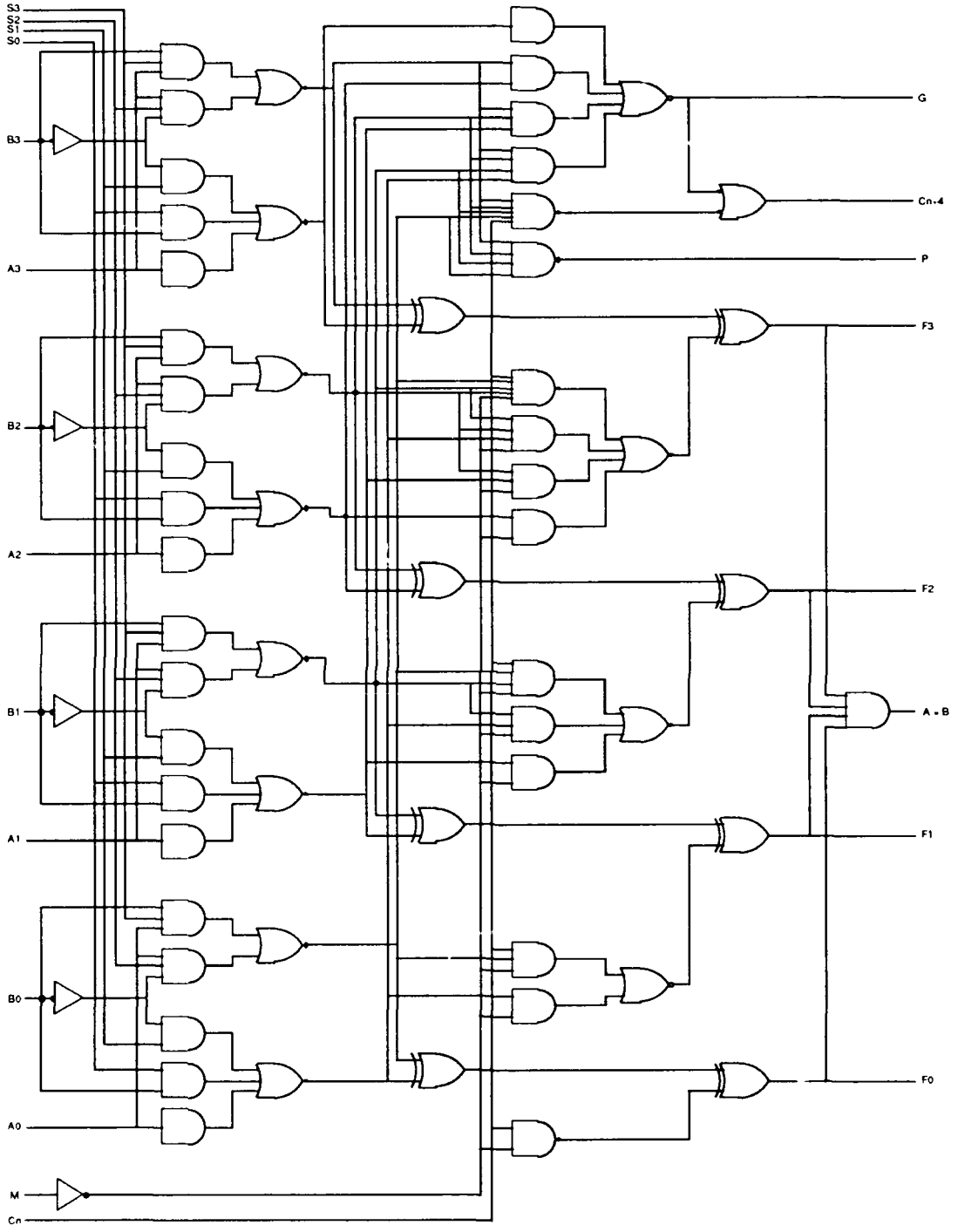


Figure 16: *Logic diagram for the 74181 four-bit arithmetic logic unit (ALU) (Adapted from Ref. [18]). There are 63 logic gates.*

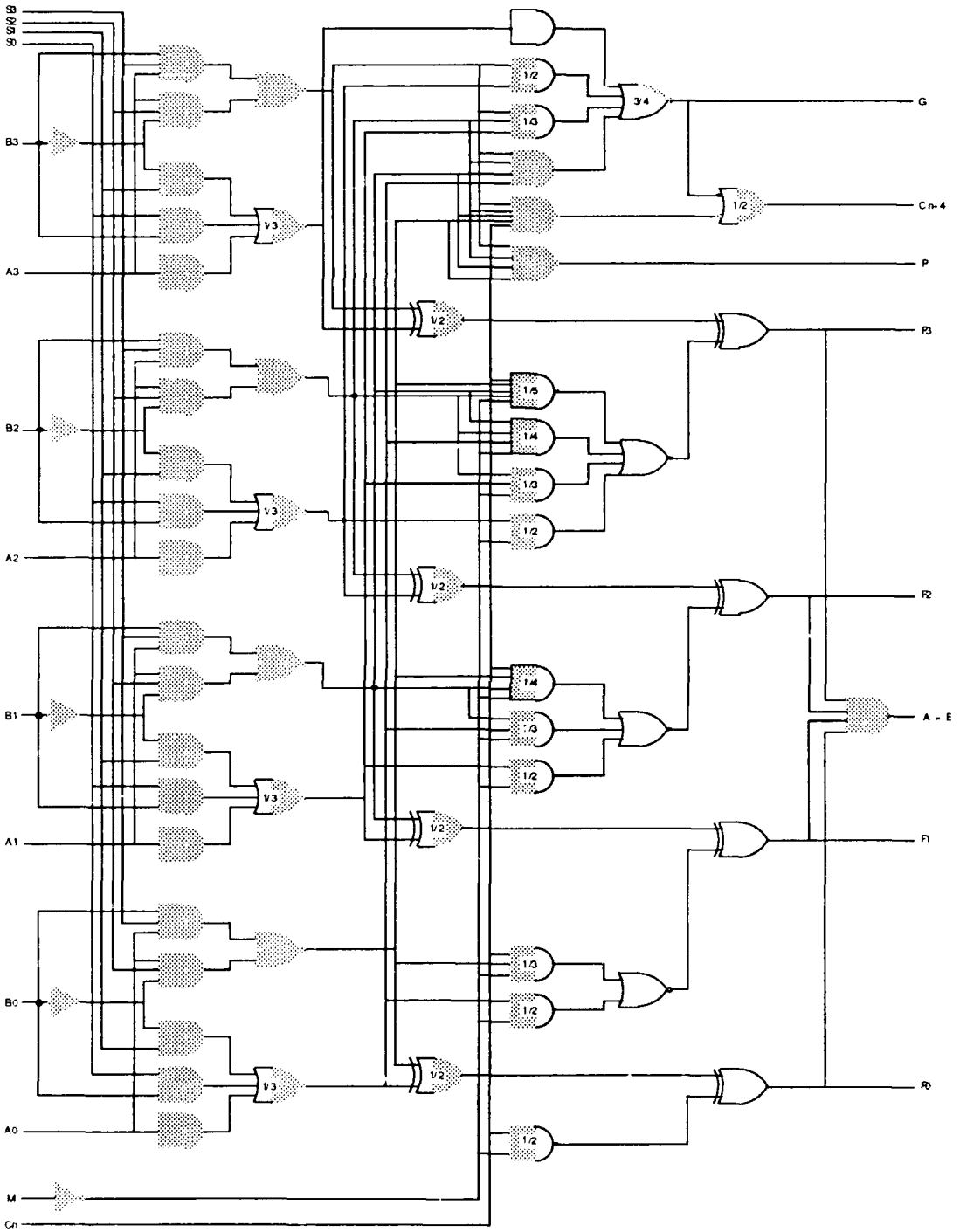


Figure 17: *Modified version of the 74181, showing the amount of underutilized logic (indicated by shading). Fractions indicate the percentage of gate inputs that are used for the ADD operation.*

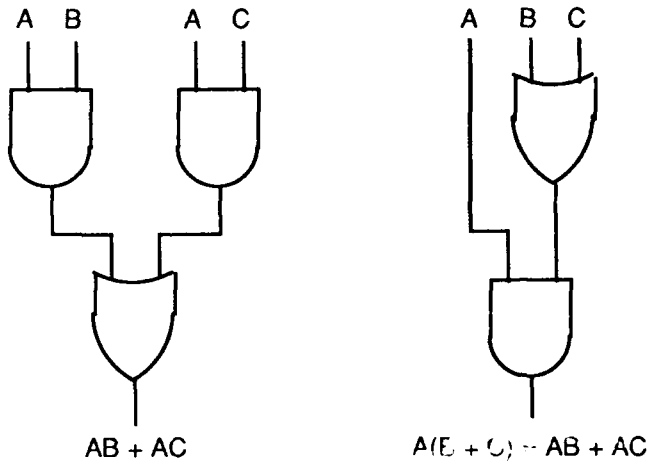


Figure 18: An AND-OR implementation of AB + AC requires three logic gates, but an OR-AND implementation requires only two logic gates.

2.6 Discussion

The fixed control sequence model is of special interest because there is no need to speculate on how the process of generating the control sequence affects performance. The control sequence is generated external to the system, prior to execution of the program, so that the mechanism that generates the sequence has little influence on the size or speed of operation of the active logic. The models described in Sections 3 and 4 are not as simple to characterize since the control sequences are generated within the processing loops of the running systems.

The fact that branching object code can be supported with a fixed control stream is a significant observation. A typical claim is sometimes made that a conventional central processing unit (CPU) can be emulated with a smaller unit that is time multiplexed to achieve the functionality of the larger unit, at the expense of greater time. This is a common textbook style of tradeoff encountered when trying to squeeze an entire CPU onto a single integrated circuit (IC). What is surprising here is that this tradeoff can be made without sacrificing performance when done properly. The underlying phenomena that makes this possible is the ability to modify the interconnects at the bit rate, such as through reprogrammable masks. This result is significant because it means that conventional general purpose computing can benefit from reconfigurable interconnects, as opposed to just special purpose applications, which has traditionally been the domain of optical computing technology. Further, the only portions of a computer that have a real need for this level of technology are places where there is a critical time/space tradeoff, such as in the CPU. Components that are normally partitioned into separate units rather than being squeezed onto one chip, such as main memory or input/output (I/O) interfaces, have little to gain from this approach since other problems such as gate speed and memory bandwidth limit performance. Thus, even

though the reconfiguration technology may be very costly initially, only a small portion of a conventional computer needs to use it in order to be effective.

Since only a small portion of a conventional computer has a need for this technology, the inefficiencies involved in discarding light through a reprogrammable mask approach might be tolerated, since the potentially more efficient beam-steering approach is too slow for this one application. However, in practical terms, the beam-steering approach is currently more easily demonstrated.

3. Runtime selected control sequences

For runtime selected control sequences, the order of execution is not determined until the time of execution, since the ordering depends on data and intermediate results. For this model, a fixed number of control sequences are stored externally and are imaged into the system on demand.

3.1 The runtime selected control sequence model

The runtime selected control sequence model is composed of a logic stage and an interconnect stage arranged in a loop, with a sequence selection mechanism included in the path between the system output and the interconnect control input, as shown in Figure 19. Control can be performed either with reprogrammable masks or with beam-steering elements, for example. Results of previous computations influence a sequence selection mechanism, which chooses the next control sequence.

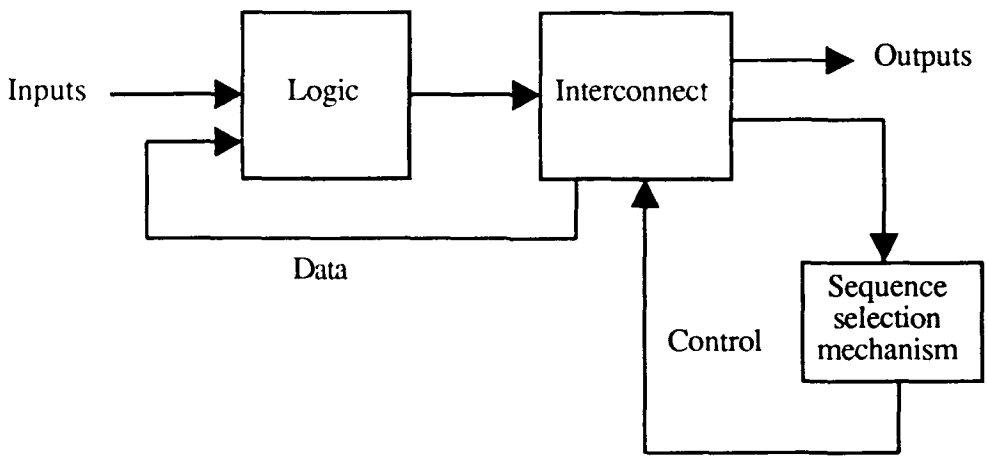


Figure 19: *Block diagram of the runtime-selected control sequence model. Results of computation influence a sequence selection mechanism, which chooses the next control sequence.*

3.2 An example: the pivoting problem

A potential opportunity for the runtime selected approach is the application of Gaussian elimination to the solution of linear equations (see Appendix C). The process is decisionless if the problem of pivoting is ignored. However, in the real world, zeros or very small numbers do in fact appear on

the diagonal, so that the pivoting problem must be addressed. A major difficulty with pivoting is that fast Gaussian elimination implementations make extensive use of pipelining, and the pipelines must be flushed whenever a pivoting decision is made. The problem grows as the size of the matrix increases, since larger columns must be scanned for the pivot element, and since more time is needed to interchange larger rows. One approach that eliminates the larger row problem uses pointer indexing. Whenever a matrix element is accessed, a lookup table maps the requested address to the real address, so that a row interchange requires only a pointer interchange, which is a constant cost regardless of how large the matrix grows. A problem with this approach is that every access to the memory goes through the address mapping mechanism, which increases the cycle time and decreases performance.

A reconfigurable interconnect technology can offer a solution without compromising performance significantly by simply reconfiguring the decoder tree of the memory to appear as if data has been moved. For example, consider the augmented matrix shown in Figure 20. The indices in the upper left corner of the 12 cells indicate the addresses of the memory locations that store the corresponding coefficients.

0 a_{00}	1 a_{01}	2 a_{02}	3 b_0
4 a_{10}	5 a_{11}	6 a_{12}	7 b_1
8 a_{20}	9 a_{21}	10 a_{22}	11 b_2

Figure 20: *An augmented matrix is shown for a system of linear equations in three unknowns. Indices in the upper left corners of cells indicate the storage locations in a random access memory.*

Figure 21 shows two decoder circuits that map four-bit addresses into spatial locations. The circuit on the left is a generic decoder that maps addresses into locations according to the matrix layout shown in Figure 20. The circuit on the right of Figure 21 shows the configuration of a decoder that swaps the top and bottom rows of the matrix. Notice that the actual data has not moved, and that there is no need for an additional address remapping lookup table that is used in a pointer approach. Only 1/6 of the crosspoints are changed between the two forms, even though 3/4 of the elements are interchanged. Further, for a modest word size of 32 bits, the total number of bits that are interchanged are $3/4 \cdot 12 \cdot 32$ bits = 288 even though only 16 bits are changed in the decoder. An important property of this method is that the modified decoder is simply projected into the system without regard for the actual data being interchanged. Thus, explicit datapaths do not need to be constructed among the rows of the matrix in order to implement the interchange. This property is more greatly pronounced for more complex interchange operations such as a transpose, in which every element is affected. A single precomputed decoder is all that is needed to

implement the transpose. Thus, the performance advantage of using pointer addressing can be achieved without paying the increased delay penalty that is typically associated with an electronic implementation.

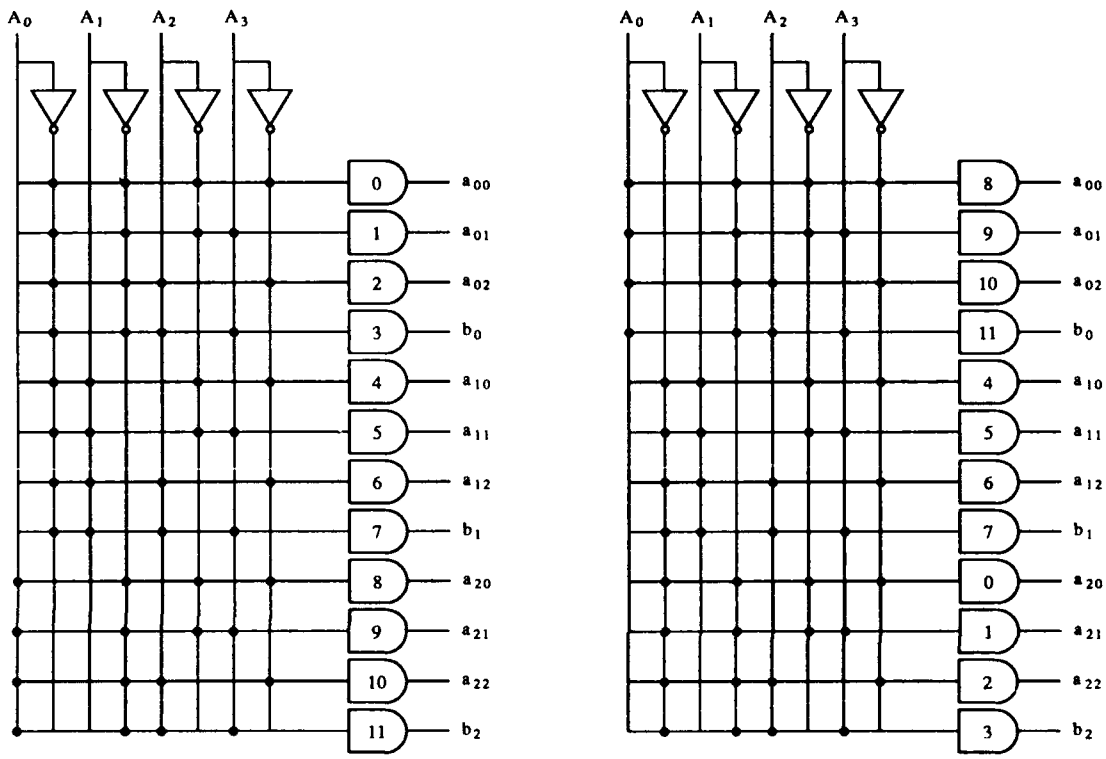


Figure 21: *Two forms are shown for a four-variable address decoder. The configuration on the left corresponds to the cell numbering shown in Figure 20. The configuration on the right corresponds to a cell numbering where the top and bottom rows are interchanged. Only 1/6 of the crosspoints are changed between the two forms, even though 3/4 of the elements are interchanged.*

3.3 Hardware caching

Observations that most of the instructions that are executed on general purpose computers are simple (such as MOVEs) and involve few stack manipulations for subroutine linkage [19] motivated the development of reduced instruction set computers (RISCs) such as the SPARC and RISC II. Another observation is that approximately 90% of the execution time of a program is spent in just 10% of the code, and that only a fraction of the 10% is active for a given interval of time. This observation led to the development of cache memories, in which the small fraction of code that is active is kept in a small fast memory that is local to the processor. The result is an overall improvement in performance.

Besides temporal and spatial locality which apply to memory, there is also functional locality which applies to a processor. A hardware cache is a novel concept described here that has no apparent counterpart in conventional digital electronics. A hardware cache keeps the most recently used hardware in a high speed, low latency section of a processor. Some of the work reported by Hennessey and Patterson [19] provides quantitative evidence that some applications use only a subset of the available instruction set of a processor. For example, a numerical based simulator such as SPICE makes heavy use of floating point operations, whereas a symbolic application such as the TEX text formatter makes heavy use of conditional branches and does not make use of floating point operations. Thus, for any given application, it is likely that only a portion of the architecture will be utilized, which creates an opportunity for a hardware cache mechanism that maintains the most frequently used portion of an architecture in a high speed, low latency section of a processor. In order for this to be effective, a study is needed on the relative frequency of instruction execution with respect to the number of different instructions needed for a given time interval. The time constraints of the Phase I effort did not allow for further investigation in this area, but an investigation into this area is proposed for a Phase II follow on effort (see Section 7.1).

3.4 Discussion

The runtime selected control sequence model appears to offer the greatest improvement in performance when the interconnects only need to be reconfigured occasionally since there may be a substantial latency between the time that a decision is made to reconfigure and the time that reconfiguration is completed. This holds true for special purpose applications such as the Gaussian elimination problem discussed earlier as well as for hardware caching. The requirement that reconfiguration must be an infrequent occurrence does not present a major problem in general. An example of an application that favors this requirement is fault tolerance, in which the interconnects are modified to bypass hardware failures that occur during operation. This is discussed in greater detail in Section 5.1.

4. Runtime generated control sequences

It may be the case that the form of a computation is not known until runtime. For example, for a phased array radar application, Gaussian elimination may be applied to the problem of solving systems of linear equations. It is a relatively straightforward problem for a computer designer to trade the precision of computation for the size of the matrix to be solved. However, once the system is designed and implemented, there is little opportunity for a subsequent change during operation in the field. Further, it is unreasonable to design all anticipated tradeoffs into the system and allow an operator in the field to select the needed configuration, since the amount of hardware needed is prohibitively large. An alternative is to allow tradeoffs to be made in the field via software, but context switching is a time consuming process when done under programmed control rather than with dedicated hardware. Thus, there is an opportunity for a mechanism that generates new hardware during the course of computation based on changing needs of the computation.

4.1 The runtime generated control sequence model
The runtime generated control sequence model is illustrated in Figure 22. The results of running computations influence a mechanism that generates new hardware during the course of computation, at the expense of some other expendable hardware.

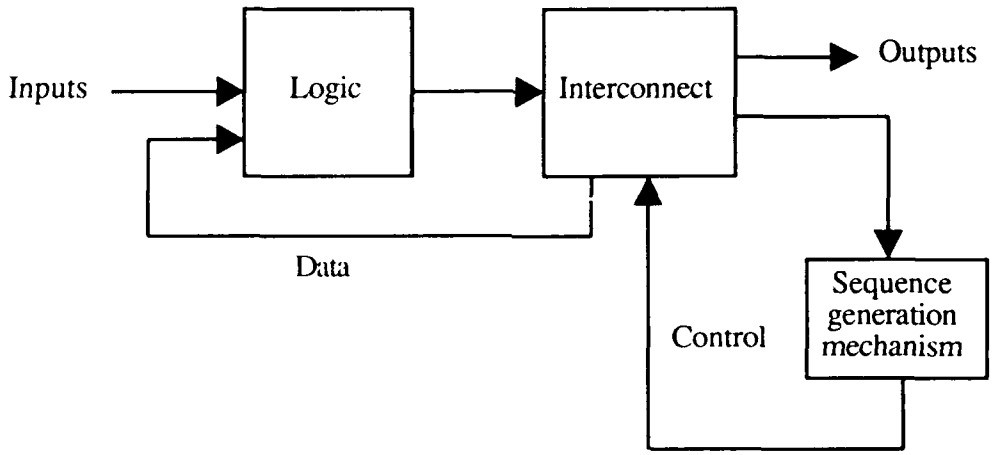


Figure 22: *Block diagram of the runtime generated control sequence model. Results of computation influence a sequence generation mechanism, which creates the next control sequence.*

4.2 An example: a runtime generated floating point processor
The purpose of using a floating point representation in a digital computer is to trade precision for range, so that fewer bits are used to store numbers. This results in a space savings for storing data and in a space savings for arithmetic logic, at the expense of precision of the data. In general, the error introduced in using a floating point representation is not a major problem as long as the computer programmer understands the limitations. An example floating point format is shown in Figure 23. There is a sign bit, a four-bit exponent, an implied radix point, and an eight-bit mantissa. If a base of two is assumed, then the bit pattern 0001011000000 represents the decimal number $.75 \times 2^2 = 3$. If the same bit pattern is used but the base is changed from 2 to 16, then the decimal equivalent is $.75 \times 16^2 = 192$. Thus a change in the base increases the range, but the precision is reduced since the number of bits in the mantissa has not increased relative to the range.

A widely used floating point format is the IEEE 754 standard. The use of a standard simplifies design decisions in creating computer architectures and in creating high-level compilers, but the generality also compromises efficiency and may even introduce significant errors when a programmer is unaware that the floating point format and the application being programmed are not matched. Consider that compilers for the C programming language, which is commonly used in the scientific community, do not generate code that reports back to the user on overflow or underflow conditions, although they do generate code that reports back for division by zero. The reason that C compilers do not automatically insert code for error checking is that the execution of

the program would be hopelessly bogged down in checking. Even if checks are made, overflows may not be recognized when special algorithms are used, or they may be recognized when they are not intended. Although these checks can be made in hardware, the typical result of recognizing an exceptional condition is to report back to the user, which thrusts the burden of dealing with the details of the floating point system back on the user. A computer program can be created in such a way that roundoff errors, overflow errors, and underflow errors are recognized and are dealt with automatically. Conventional approaches to dealing with these errors are to either ignore them or to inform the user of the problem. There is no attempt to modify the floating point format to match the changing nature of the data because of the enormous complexity of doing this in software. A hardware implementation of a self-modifying floating point unit would be faster if possible, but a conventional hardwired approach is hopelessly complex since logic would have to be included for a wide range of floating point variations. Although the design of a floating point unit is not trivial, its functional behavior can be described at a high enough level that a competent designer can leave the base, exponent, and mantissa sizes as variables in the description as discussed below. The translation from functional description to physical layout can be automated, and the suggestion made here is to transfer the responsibility of that mapping from the designer to the sequence generator.

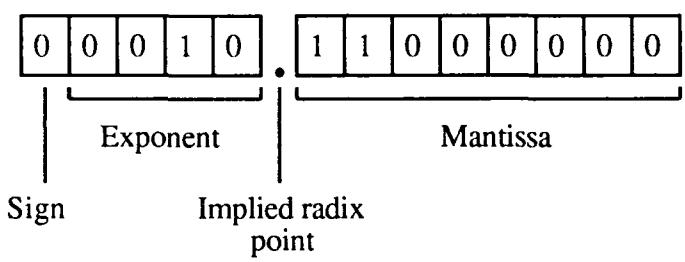


Figure 23: *An example floating point format has a sign bit, a four-bit exponent, an implied radix point, and an eight-bit mantissa. The bit pattern shown represents the number .75 x 2^2 = 3.*

A hardware compiler is a computer program that translates high level descriptions of computer hardware into physical designs. Hardware compilation is normally used in the design of VLSI circuits. A high level AHPL [20] description of an IEEE 754 floating point unit that performs addition and subtraction of single precision numbers is described below.

In the AHPL description, which is adapted with modifications from Ref. [20], the two input operands are ***A*** and ***B*** and the result is ***C***. ***_s*** is the sign bit, ***_E*** is the exponent, and ***_S*** is the significand. The DEFINE statements set the widths of the exponent, the significand, and the radix. The DEFINE statements are the entry points for a process that generates custom floating point units for radix 2 for this example, using a hidden '1' (as in IEEE 754).

```
DEFINE    EXP_WIDTH    8.    # The width of the exponent in bits
DEFINE    MANT_WIDTH   24.   # The width of the mantissa in bits
DEFINE    BASE         2.    # The radix

# Exponent calculation: A-exponent > B-exponent
1     → (CMPR(AE;BE))/(2, 7, 8).        # Compare exponents
2     CE ← ADD[1:EXP_WIDTH](AE;BE'; 1). # AE > BE
3     → (CMPR(CE; EXP_WIDTH T MANT_WIDTH))/(4, 4, 5).      # Is exponent
                                # difference > significand size?
4     C ← A;                    # Exponent difference too great, result gets larger operand
      → (28).
5     BS ← 0,BS[0:MANT_WIDTH-2]; CE ← DEC(CE).      # Shift smaller operand right,
6     → (+/CE)/(5).             # till exponents agree. Degree of shift depends on BASE
7     CE ← AE;                  # Assign larger exponent to exponent of result
      → (14).

# Exponent calculation: B-exponent > A-exponent
8     CE ← ADD[1:EXP_WIDTH](AE';BE; 1). # BE > AE
9     → (CMPR(CE; EXP_WIDTH T MANT_WIDTH))/(10, 10, 11).   # Is exponent
                                # difference > significand size?
10    C ← B;                    # Exponent difference too great, result gets larger operand
      → (28).
11    AS ← 0,AS[0:MANT_WIDTH-2]; CE ← DEC(CE).      # Shift smaller operand right,
12    → (+/CE)/(11).            # till exponents agree. Degree of shift depends on BASE.
13    CE ← BE;                  # Assign larger exponent to exponent of result

# Perform addition or subtraction on significands
14    AS * as ← ADD[1:MANT_WIDTH-1](AS'; 0; 1).      # If sign bit is negative, form
                                # 2's complement of A
                                # If subtracting and B is negative, 2's complement of B
15    BS * (bs XOR subf) ← ADD[1:MANT_WIDTH-1](BS'; 0; 1).
      bs * subf ← bs.     # Change sign of subtrahend when subtracting, so that 2's comp of
                          # B addition is used for addition
16    cfp,cs,CS ← ADD((as,AS);(bs,BS)).     # Perform the ADD, set overflow bit cfp
```

```
17    → (as XOR bs, as ∧ bs, as' ∧ bs') / (20,18,19).        # Check signs
18    cs,CS ← ADD[1:MANT_WIDTH-1](cs',CS'; 0; 1).   # Convert from 2's comp
                          # negative to sign+mag
19    cs,CS ← cfp,cs,CS[0:MANT_WIDTH-2];  CE ← INC(CE).  # Shift right to
                          # compensate for overflow. This varies according to BASE.
20    eof * (∧/CE) ← 1.  # Check for exponent overflow
      → (28).

# Normalize result. Check for true zero. Restore signed magnitude for negative number
21    CS * cs ← ADD(CS'; 0; 1). # If result should be negative, then restore to negative
                          #If result is true zero, set exponent to zero
22    CE * ~(+/CS[1:MANT_WIDTH-1]) ← EXP_WIDTH ⊤ MANT_WIDTH.
      → ~(+/CS[1:23]) / (28).
23    → (CS[0]) / (28).                  # Exit if result is normalized
24    CS ← CS[1:MANT_WIDTH-1],0; CE ← INC(CE).      # Shift significand left.
                                         # Degree of shift depends on BASE.
25    → (+/CE)/(27).                     # A zero exponent indicates underflow
26    CS ←  MANT_WIDTH ⊤ 0. # If result should be negative, then restore to negative
      → (28).
27    → ~(CS[0]) / (24).                 # Normalized?  If not, go back to 24.
```

The generic floating point architecture that a hardware compiler would produce takes the same form, such as shown in Figure 24, regardless of the choice of exponent, base, or precision, however, the widths of the channels and the supporting logic will change in accordance with the AHPL description. The channel widths are indicated in bits by the numbers adjacent to the diagonal slashes.

4.3 Discussion

The runtime generated control sequence model represents a major departure from conventional computer architecture. The reconfigurable floating point processor is a simple example of this concept, but a still greater application may be found for the dataflow paradigm, which theoretically supports maximum parallelism, by tailoring the architecture to suit the problem. This is discussed in greater detail in the next section.

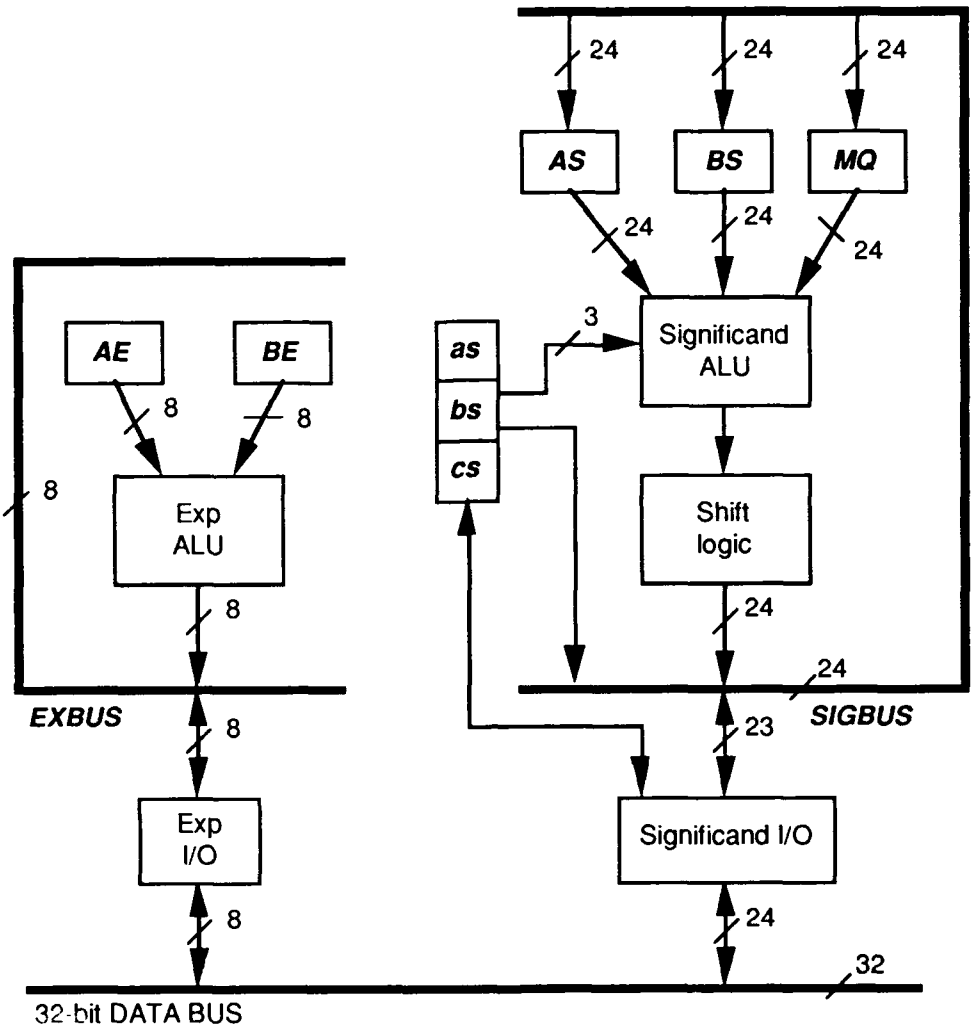


Figure 24: *Basic block diagram of a floating point coprocessor showing an exponent processor (left) and a significand processor (right). (Adapted from Ref. [20]).*

5. Applications

Four applications are described here that are particularly well-suited for a reconfigurable interconnection technology.

5.1 Fault tolerance

An application of the reconfiguration technology that has special significance to the Department of Defense is fault tolerance. The general problem is for a computer to deal gracefully with a situation of compromised reliability, caused either through systemic failures (as in device burnout) or

through catastrophic failures (caused by fires, shock, *etc.*). The fault tolerance problem can be partitioned into subproblems as shown below:

- Detection
- Isolation
- Propagation
- Repair
- Recovery

The sequence of events that occur during a failure begins with the detection of the fault. The fault is then isolated to a particular system, and is propagated from within the logic to a known location. A repair mechanism is then invoked, and then finally, a recovery is made so that no corrupted data is introduced into the system.

A simple redundancy scheme is used here, which is typical for systems that need to tolerate failures in the active switching elements.[5] The triple modular redundancy [21] fault tolerant model is shown in Figure 25.

Fault detection is accomplished by a voting mechanism that compares the results of three identical systems operating on identical sets of data. For triple modular redundancy, the systems compare their states with the others, and resynchronize when there is a disparity, which addresses both the transient fault and the stuck-at fault problems. Note that the fault cannot be isolated to a particular system with a voting strategy if fewer than three systems are used. The fault mechanism is invoked only when a fault occurs, which is a statistically rare event. Thus, systemic failures resulting from device burnout, thermal breakdown, defect migration, *etc.*, are a more remote possibility for the fault tolerance mechanism since it is used infrequently. Note that the fault tolerance problem for optical computing with free-space interconnects is simplified since there is virtually no coupling between optical logic gates, and since it can be assumed that masks, lenses, and free space suffer no systemic failures.

The focus here is on the repair mechanism, which may involve some level of redesign. Once a fault has been detected, the system can continue error-free operation as long as another fault is not introduced into one of the remaining operational systems. The interval between failures can be used to redesign the failed system to bypass the fault, which can either be done manually or automatically depending on the time scale involved, which effectively restores the system to the original fault-free state. The implementation of a redesign process is a potential opportunity for

[5]There are simpler fault recovery mechanisms that deal with transmission and data storage errors, that use cyclic redundancy codes or Hamming codes [21], for example. These traditional methods apply here with equal effectiveness, however, they do not offer a solution to the more difficult failures associated with active logic faults, which is addressed here.

using a slow beam-steering mechanism since reconfiguration to bypass faults is an infrequent event.

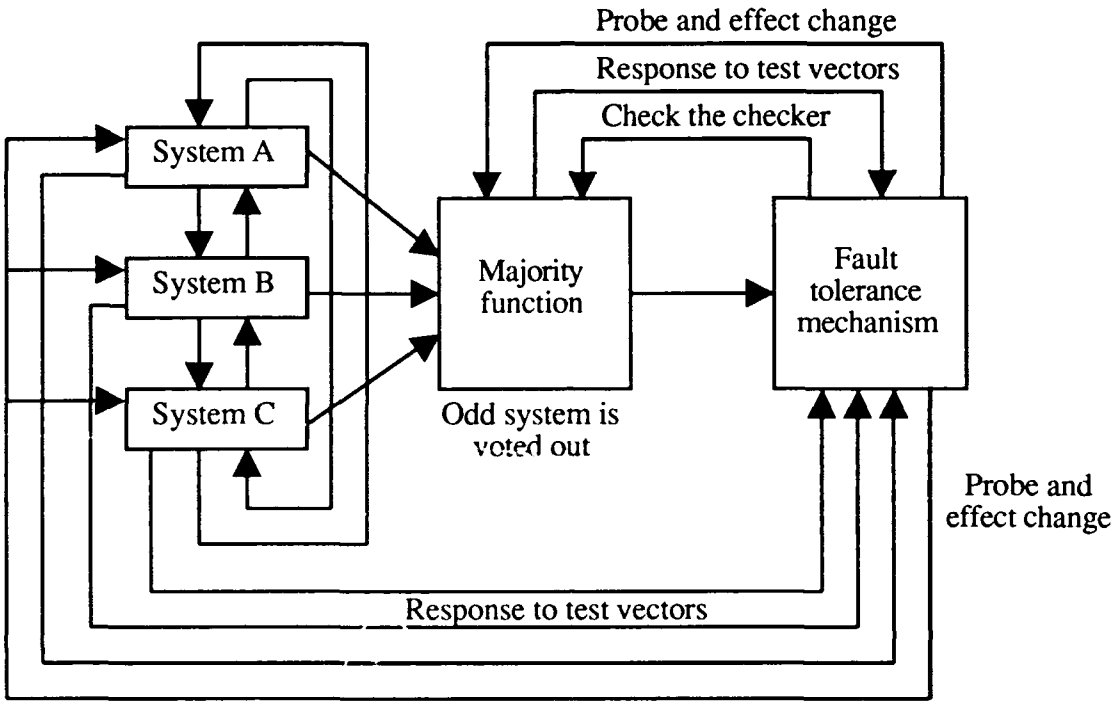


Figure 25: *Three systems A, B, and C send their results to a voting mechanism (the majority function) that determines if a fault is present, which then invokes the fault tolerance mechanism.*

5.2 Parallel processing

In this section, the importance of interconnection networks in parallel processing is discussed, and the claim is made that an optical reconfiguration technology can improve performance. Section 5.2.1 shows that an ideal network, such as a completely connected graph, simplifies programming and guarantees an improvement in performance. Unfortunately, a completely connected architecture with a large number of processors is impractical to build using electronics technology. As a result, most of the parallel architectures that have been constructed try to approximate a completely connected architecture using other technologies such as adding communication co-processors (as in the Intel, DELTA, and nCUBE-II machines), fast communication busses (as in the Alliant machines), parallel I/O (as in the Tera computer), switching networks (as in the Masspar and BBN butterfly) and so on. Almost all electronic solutions have limitations which are discussed here. A solution that makes use of optically reconfigurable interconnects provides a potentially significant alternative for simulating a completely connected architecture. The conclusion is made that a hybrid architecture consisting of electronic processors with an optically reconfigurable

network may provide an excellent alternative solution to the communication problems associated with all-electronic implementations.

The Problems: There are three fundamental problems in parallel processing that are considered here:

- detection of parallelism, given an algorithm specified in a high level language;
- partitioning the parallel program for a given architecture;
- mapping the program onto the architecture and producing a scheduling of tasks to processors.

Consider the following program that represents a matrix times a vector $\mathbf{y} = A\mathbf{b}$ algorithm:

```
v[1:m], A[1:m,1:m], b[1:m]

for i=1:m
    v(i) = Σ A(i,1:m)*b[1:m]
endfor
```

The parallelism inherent in this program can be expressed with a dataflow graph, such as the fork directed acyclic graph (DAG) shown in Figure 26.

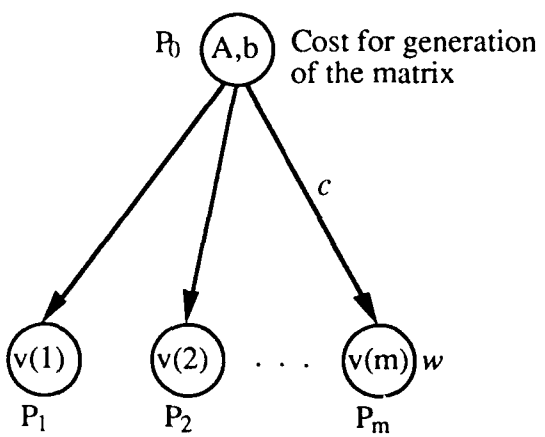


Figure 26: A *dataflow graph represents matrix-vector multiplication.* c *is the cost of communication and* w *is the cost of computation.* P_i *represent tasks.*

Now given an architecture, the above computation must be partitioned so that the execution time is minimized. The partitioning problem depends on the interconnection network, the communication speed, and the computation speed. The general partitioning problem is extremely difficult (NP-complete) even for a simple architecture that is completely connected. As a result of this difficulty, many approximate solutions have been proposed for the partitioning problem, both in software and hardware.

Now consider associating communication and computation weights with the above task graph. The cost for the multiplications is w and the cost for communication is c for reading a row of A and b. An important quantity regarding the partitioning of a task graph is the granularity, which is the ratio of computation to communication. For the above graph this ratio is:

$$g = \frac{w}{c}$$

Depending on the granularity g a decision can be made if tasks will be executed sequentially in the same processor or in parallel. For example, if an architecture has m processors and is completely connected, and if $w > c$, then the optimum parallel time is:

$$PT_{opt} = c + w + \text{cost for generation of A,b}$$

and is obtained by executing every task in a separate processor. In other words, the optimum partitioning is achieved when each task is mapped to a separate processor.

For a counter example assume that the communication cost $c > mw$. Then the optimum partitioning maps all tasks and the matrix generation onto one processor since

$$c + w + \text{cost for generation of A,b} > mw + \text{cost for generation of A,b}\ .$$

The above example demonstrates that communication and computation parameters are important in deciding how program execution should be partitioned in parallel architectures. The example also illustrates the importance of granularity in computation. Large granularity implies better system utilization, and as a consequence, better performance as well. In a related Rutgers effort [22,23] it is shown that fast algorithms for scheduling and program partitioning can be created with a guarantee of a factor of two in performance when compared with the optimum for coarse grain parallelism. A nearly maximum system performance can be achieved whenever the granularity is greater than one.

A problem in achieving the maximum system performance is in how to increase the granularity. Considering the above example, one solution is to increase the communication speed with respect to computation, so that w/c increases. This approach has led to numerous solutions for increasing the communication speed of electronic computers, as described below:

- the addition of cache for uniprocessors, which increases granularity by reducing communication;

- the creation of faster busses for shared memory parallel architectures;

- the addition of communication co-processors or switching networks for message passing architectures (such as the Intel, nCUBE, Masspar, and Connection Machine computers);

- the creation of custom architectures for special programs, such as systolic array architectures for the matrix multiplication example above.

All of the hardware solutions described above have limitations:

- Cache misses can deteriorate performance unless the mapping of data to memory is suited for the given program.

- Fast busses are extremely expensive to construct, but even if faster busses are created, they still pose an obstacle since busses in general sequentialize communication.

- Dynamic switching networks are a compromise solution to a completely connected architecture. Switching networks reduce the traveling distance for communication but the cost of changing the switch settings must be added into the communication cost. Thus, performance depends on the number of times that the switching network is modified during execution.

  The use of communication co-processors reduces the transmission speed between distant processors. For example, the Intel-II and nCUBE-II architectures use communication co-processors, and the resulting communication between distant processors is almost the same as communication between neighbors (within 99%) if the communication load is low. However, as the communication load increases, communication deteriorates significantly.

- Special architectures perform extremely well for special problems. For example, systolic architectures perform well for matrix multiplication and signal processing problems, but performance is poor for problems that are not suited for such architectures.

The above hardware solutions have the common goal of increasing the granularity by reducing the traveling distance of the data. The resulting architectures approximate completely connected architectures. Unfortunately, the limitations mentioned above are inherent in the architectures and are technologically very difficult to overcome with electronics. Evidence that a reconfigurable

interconnection technology may improve the performance of parallel processors is summarized below:

No MIMD shared memory machine with 1000 processors has been built as of yet and it is not clear if it is possible to build such a machine at a reasonable cost. The DARPA supported Tera Computer project has a goal of 256 processors at an estimated cost of 30 million dollars (source: Burton Smith, president of Tera Computer.)

Message passing architectures have been built with thousands of processors, such as the nCUBE-II with 8K processors, and the DARPA/Intel supported Touchstone project with 1024 i860 processors. Both of these new message passing architectures have a static interconnection network that uses co-processors for communication. A major disadvantage is that programming such architectures is extremely difficult, particularly since a heavy communication load makes an estimation of the cost of architecture parameters difficult. No automatic scheduling tool exists for such architectures and programming must still be done manually. As discussed above, only for an ideal architecture such as a clique (a completely connected architecture), can programming that guarantees performance within factor of two of optimal be possible for coarse grain parallelism.

Another disadvantage of current electronic approaches to parallel processing is that electronic communication technology does not keep pace with processor technology. For example, in going from the iPSC/1 to iPSC/2 the processor speed increases by a factor of seven while communication speed between two neighbors increases only by a factor of between two and three. This lag in technology results in a decrease in granularity, which reduces performance for systems with large number of processors.

Switching networks are attractive for communicating among processing elements because they can be reconfigured to map a task graph onto the architecture. A static reconfiguration implies that once the architecture is reconfigured it does not change during execution. As discussed above, the performance of a switching network depends on the frequency of switching and its impact on the granularity. A disadvantage of an electronic switching network is that messages pass through the switches. This poses a problem in the BBN Butterfly based network for example, when two messages arrive at the same switching node and require the same output port. The messages are then sequentialized.

To summarize, creating a network that perfectly maps a task graph onto a multiprocessor has numerous advantages (for a perfect mapping of a task graph to an architecture graph the cardinality[6] of the graph is minimized):

---

[6]Cardinality as used here refers to how closely the graph of the computation matches the interconnection graph of the target machine.

• A software advantage to using a reconfigurable network is that compilers can be written that are fast and achieve good performance. One such compiler is under development at Rutgers which is based on the macro-dataflow model of computation using a completely connected architecture. A prototype has already been built which runs on existing architectures such as the nCUBE-II and the Intel hypercubes.

• Because the mapping is optimal no data messages are sequentialized in the network. Avoiding such delays results in better architectural parameter estimates and better compiler performance.

• The communication cost decreases and the granularity increases. This is necessary to take advantage of the new superfast processors such as the Intel i860 rated at 60 MFLOPS for single precision arithmetic. To build parallel architectures with large numbers of processors, for example 1000 processors using the i860, a superfast network is needed, otherwise the throughput of such an architecture is compromised.

5.3 Dataflow computing

In the von Neumann model of a digital computer, instructions are executed in sequence as they are read from memory under the control of a program counter. Successive instructions reside in successive locations in memory except when there are branch instructions, which make temporary changes in the sequence. The traditional model fetches an instruction, executes it, and repeats this process until completion of the program. This single fetch-execute sequence (or thread) is the well known von Neumann bottleneck. There has been considerable interest in computer architectures which execute more than one instruction at a time, and with the arrival of VLSI circuitry, many different architectures have been proposed and built which execute multiple instructions in a single fetch-execute cycle. The success of these architectures depends on the applications and on the ease of use. As one might expect, those machines with relatively limited parallel operations, or parallel machines built to handle specific applications, have been most readily accepted. The latter has enjoyed acceptance because the performance improvement can easily be demonstrated; the former has enjoyed acceptance because it has proven simpler to modify existing programming languages and programs to take advantage of modest parallelism.

With easy access to custom VLSI, there has been a rapid growth in the variety and size of highly parallel (*i.e.* lots of processors) computers which have been proposed, studied, and built. Unfortunately, it has proven easier to create large amounts of computing power than to make use of it. Most of the earlier approaches to parallel architectures assume that if enough computing power is made available in a 'reasonable' form, users will figure out how to make use of it, and indeed much effort has been made in that direction, but it has proven very difficult to detect much of the potential parallelism in an existing program (although there has been a large effort, and some reasonable success [24, 25]).

A class of architectures and programming paradigms, dataflow, exists which automatically detects parallelism, and can schedule the use of computing resources to take advantage of the parallelism. The basic model is very simple, but proposed implementations have proven far more complex [25-29].

The von Neumann model may be described as control-driven. The execution of a program is controlled by a program counter. The dataflow model of computing may be described as data-driven. In dataflow, an instruction executes if:

- there is a resource available to execute it, and
- all of the operands required are available.

The dataflow model of computing is unique in the sense that it theoretically supports maximum parallelism.

> "The key idea behind a dataflow system is that there are no variables in the usual sense, that is, no addressable words that can be read from and stored into. Instead, values are represented by packets that are transmitted between processing units. Each processing unit has the task of computing some function of its inputs in the strict mathematical sense: each function depends only on its inputs, and not on any global variables or other side information. Furthermore, the only thing a function does is produce an explicit result. It has no side effects such as modifying global variables, because there are none.
>
> Because there are no variables or side effects, each processor may begin its computation as soon as its input packets have arrived. There is no program counter and no explicit sequencing of computations, other than that implicit in one calculation depending on the result of another one. For example, if $y = f(x)$ and $z = g(y)$, then f must run before g. It is precisely this ability of dataflow systems to operate without any artificially forced sequencing that makes them so attractive for distributed computing."
>
> - [Ref. 30]

A sample program is shown below that solves an ordinary differential equation that is taken from Ref. [14]. The program as written is sequential, however, a compiler for a dataflow machine can produce a dataflow graph which identifies maximum parallelism based on data dependencies and flow-of-control dependencies, as shown in Figure 27. Each of the 16 boxes represents a primitive operation, and the lines with arrows represent dependencies. If a special purpose architecture is being considered, then the dataflow graph can be used as a guide in creating dedicated hardware to implement the program. For a more traditional electronic implementation of the dataflow model, a general architecture is created that implements the dataflow graph. The generality is indispensable

because without it, computer programmers would be forced to create a new piece of hardware to solve every task. On the other hand, the generality sacrifices performance, sometimes significantly. This is one major reason that the dataflow concept is not in general use today, even though it theoretically supports maximum parallelism.

```
{Solve the differential equation y'' + 2xy + 2y = 0.}
program diffeq (input, output);
var a, dx, x, u, y, xl, ul, yl: real;
begin
        {read in the ordinate, a, at which we want the value of the function,
        the step size, dx, and boundary condition}
        read (a, dx, x, u, y);
        while x < a do
            begin
                xl := x + dx;
                ul := u - 2*x*u*dx - 2*y*dx;
                yl := y + u*dx;
                x := xl; u := ul; y := yl
            end;
      writeln (y: 13:9)
end.
```

"To see how the system of [Figure 27] operates, imagine that the input buffers of processors 1 through 5 are initialized with the values shown. Constant arguments are shown encircled; they are 'hardwired' into their respective buffers. For simplicity's sake, we will assume that each operation requires one cycle. During cycle 1, processors 1 through 5 all fire, delivering results to processors 6 through 10, all of which fire during cycle 2. Processor 9 compares its two arguments. If the first is less than the second, it outputs a packet containing a 1 (for true); otherwise, it outputs a 0 (for false). Processor 10 simply duplicates its argument. The DUP function is useful to generate extra copies of a previous result. Note that processor 5 has two outputs, but that three are needed. (Again for simplicity, we have arbitrarily limited the number of outputs per processor to two.)

During the third cycle, something amazing happens. While processor 11 is busy computing $2xu\ \Delta x - 2y\ \Delta x$, processor 5 is already busy working on the following iteration (i.e., $x = 2\ \Delta x$), even though the previous one has not yet finished. *Notice that this parallelism is completely automatic; any part of the computation that can proceed does proceed.*"

- [Ref. 14, emphasis added.]

This example is suitable for the static dataflow model, in which the form of the dataflow graph does not change during the course of computation. This behavior is not possible when there are runtime conditionals, such as for if-then constructs or for loops, which results in a more complex implementation. A simple physical (although not practically constructable) model that includes runtime conditionals might consist of:

- A set of computing resources (either a number of CPU's, or a mix of specialized resources, such as adders, multipliers, *etc.*)
- Instructions of the Form OP:A:B:C, which are interpreted as: perform Op on operands A and B and name the result C.
- An associative (content-addressable) memory for operands.

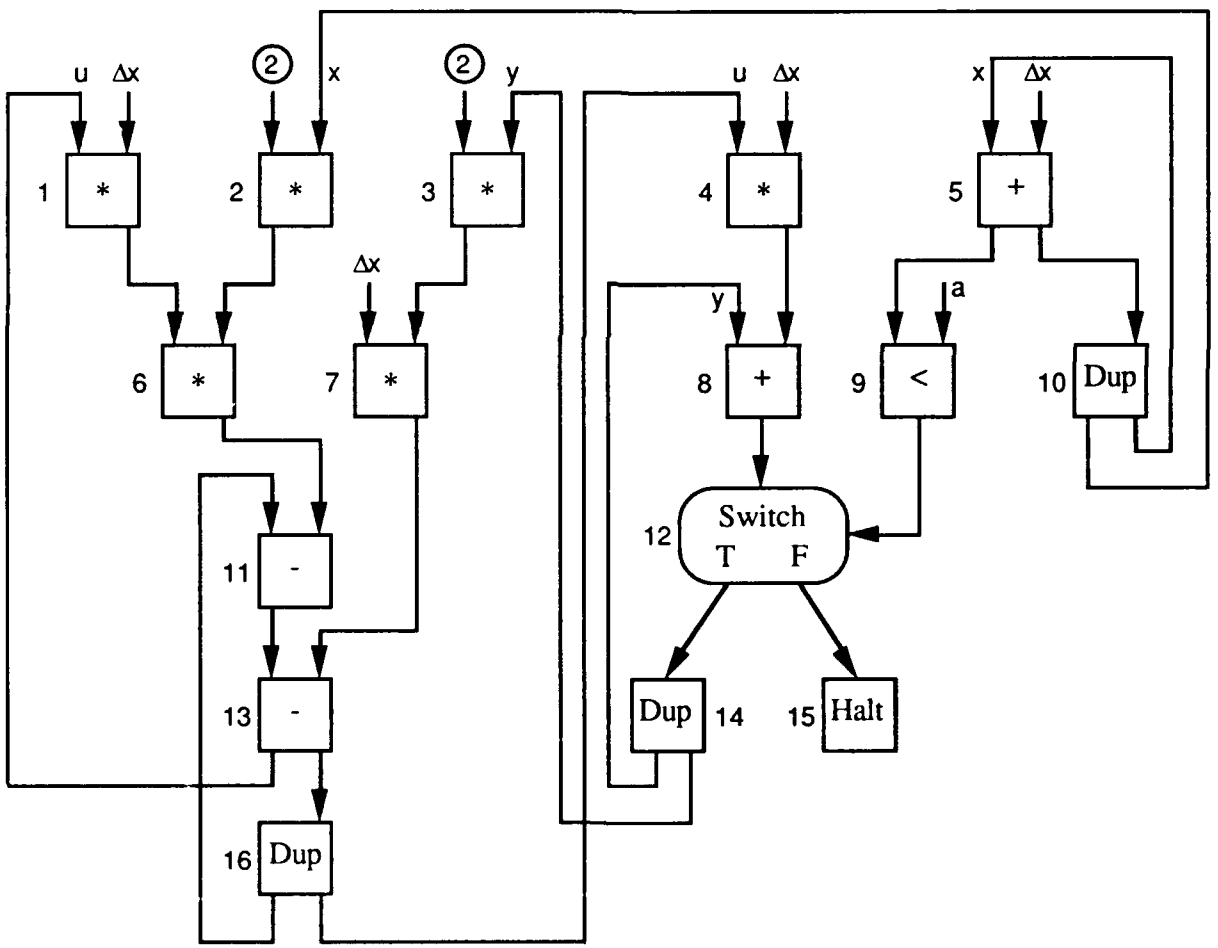


Figure 27: *A dataflow graph for solving the differential equation: y'' + 2xy' + 2y = 0 [Adapted from Ref. 14].*

The OP:A:B:C instruction cannot be executed until A and B are available. If there is a resource to do OP, it will be executed as soon as the operands are available, which can be found out by asking the content-addressable memory for C. The memory cannot deliver C until it has been created. Similarly, any instruction that requires operand C cannot execute until the current instruction executes, because its execution creates the C; C does not exist until OP:A:B:C completes. Implementations face the problem of proliferation of names (as described, each name must be unique), how to reference by name very rapidly, how to program these machines, and a host of other smaller problems.

A dataflow graph is used for the dynamic dataflow model also, where each node represents an instruction, and each edge specifies a data dependency between two instructions as above. The example used here is taken from Arvind's Dataflow work at MIT[7] [29]. The MIT project uses the language Id, designed specifically for the MIT dataflow machine. An Id program is first compiled into a dataflow graph, and then into a language which 'executes' the graph on the MIT hardware. An important point to consider for all existing dataflow computers is: *The dataflow graph does not represent a piece of hardware; it represents an algorithm.*

Consider an example of a wavefront computation, in which an n×n matrix **X**, is constructed as follows:

- *Cells along the left and top borders contain* 1, *i.e.* **X**[1,j] = **X**[i,1] = 1.
- All other cells contain the sum of their neighbors to the left and to the top, *i.e.* **X**[i,j] = **X**[i-1,j] + **X**[i,j-1].

The Id program for this computation looks like:

```
X=make_matrix (grid n) f          * this creates the empty n×n matrix
    def f(i, j) = if (i==1) then 1
          else if (j==1) then 1
          else (X[i-1,j] + X[i,j-1])
```

Consider the potential parallelism in this program, in which make_matrix initiates n^2 computations, one for each component of the array. Most of these computations must be suspended because they try to read neighboring elements that are still empty. However, all computations for the top and left border can complete immediately, since they do not depend on anything else. When the border components at (1,2) and (2,1) are filled, the computation for (2,2)

[7]From Project MAC, funded by DARPA through ONR on contract N00014-84-K-0099.

can proceed. As soon as the computation for (2,2) finishes, the computations for (2,3) and (3,2) can proceed, and so on. Figure 28 shows a snapshot of the array during the process.

1	1	1	1	1	1	1	1	1	1
1	2	3	4	5	6				
1	3	6	10	15					
1	4	10	20						
1	5	15							
1	6								
1									
1									
1									
1									

Figure 28: *Wavefront computation.*

The left and top borders and some internal elements have been computed. The shaded squares show the next components that can be computed. As shown, the computation can proceed along a diagonal wavefront that sweeps from top left to bottom right. The make_matrix statement does not imply any particular ordering on the computations for the array components. It is clear from the program that some ordering may be implied by the data dependencies. In functional languages[8] like Id all ordering is based on data dependencies. Dataflow graphs implement the notion of dynamically adjusting the order of computations to accommodate the dependencies which arise in a particular run of a program. Before looking at the dataflow graph associated with the wavefront computation, consider how the matrix is stored in a conventional random access memory, since a large content addressable memory cannot be practically built. This is a major implementation

[8]"Functional (also known as applicative) programs are built from 'pure' functions, *i.e.* mathematical functions, which in sharp contrast to functions found in conventional programming languages are side-effect free, meaning that their evaluation cannot alter the environment of the computation. In other words there is no assignable program state. Because of this we can no longer program 'by effect' so that the value computed by a program and the program itself are reduced to one and the same thing. Program execution then becomes a process of altering the form of the final value in precisely the same way that we can alter the form of '8 + 1' to '9' in the knowledge that they both denote the same value." - Anthony J. Field and Peter G. Harrison, *Functional Programming*, Addison-Wesley, (1988).

problem for dataflow machines. Assume that the matrix is stored by columns as shown in Figure 29.

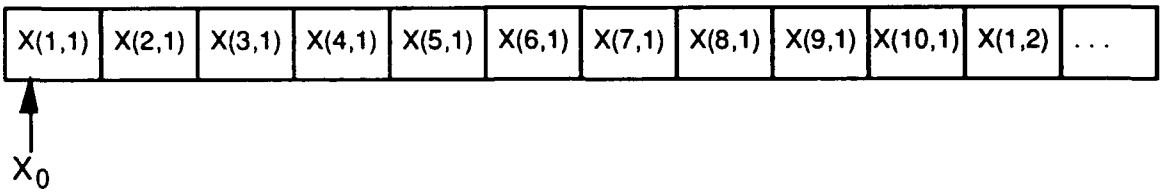


Figure 29: *Layout of array in memory.*

In order to access **X**(i,j), it is necessary to compute the address $\mathbf{X}_0$ + (i-1) + 10(j-1). After algebraic manipulation, the addresses of **X**(i,j), **X**(i-1,j) and **X**(i,j-1) are, respectively:

$\mathbf{X}_0 + i + 10j - 11$
$\mathbf{X}_0 + i + 10j - 12$
$\mathbf{X}_0 + i + 10j - 21$

With this information, a dataflow graph for the body of the wavefront computation can be constructed as shown in Figure 30.

Consider how the graph is interpreted. Each of the nodes, labeled 0 through 9, corresponds to an instruction in the computer. Any of these function nodes can 'fire' (i.e. perform its function) as soon as it has data at all of its inputs. The graph represents a partial sequencing of the instructions; that is, node 5 must fire before node 7, but it could fire before, after, or simultaneously with nodes 3 and 4.

Consider tokens (or data) placed on the entry points $\mathbf{X}_0$, i, and j. Nodes 0 and 1 can fire (computing $\mathbf{X}_0$ + i and 10j respectively). When they complete their computations, they put tokens (or computed values) on their output lines enabling node 2 which then computes $\mathbf{X}_0 + i + 10j$. Now nodes 3, 4 and 5 are enabled and can fire, computing the addresses of **X**(i,j), **X**(i-1,j) and **X**(i, j-1) respectively. Nodes 6 and 7 can fire next, fetching the values of **X**(i-1,j) and **X**(i,j-1). Node 8 adds those values resulting in the new computed value of **X**(i,j), and when node 8 fires, it enables node 9 which stores the result at the address of **X**(i,j).

In dataflow computers as they exist today, the firing process involves interrogating token (data) stores to determine which tokens are available and which functions can be performed. Much of the research is concerned with limiting the amount of memory which must be searched, and speeding the assignment of single operations to computing resources. The dataflow graph shown above represents several searches and assignments. There is no single piece of equipment which does

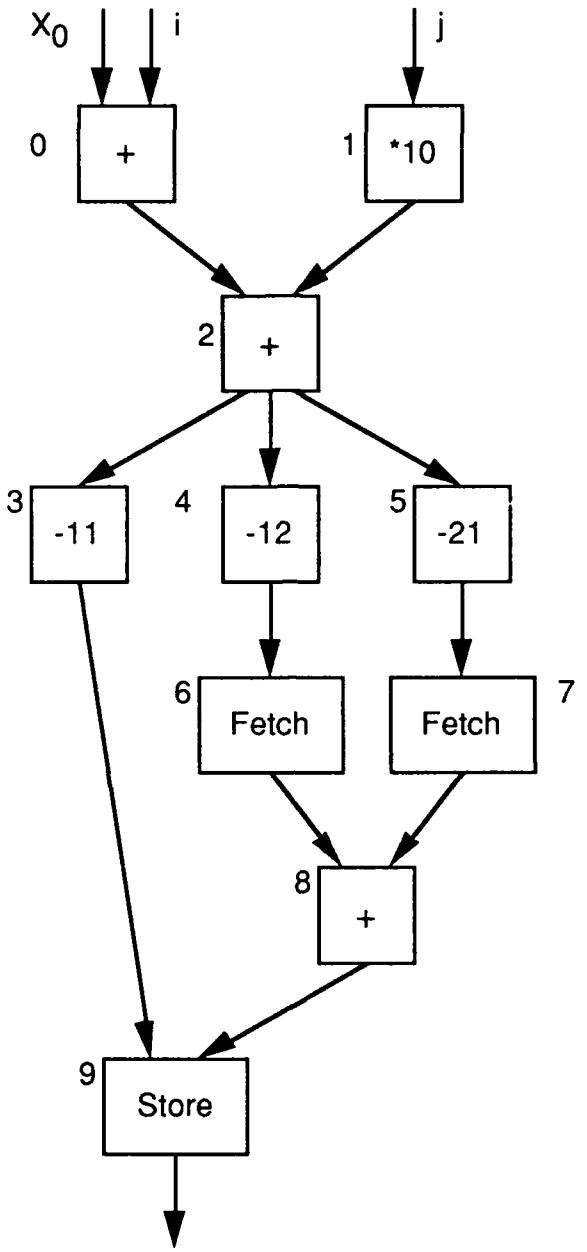


Figure 30: *Dataflow graph for body of computation.*

those operations. In the example there are many similar operations which can take place simultaneously (the wavefront diagram shows that computation can proceed at several places in the matrix simultaneously). Dataflow computers as envisioned by any of the current dataflow projects are not reconfigurable computers. These machines execute single instructions in a unit, and the

dataflow paradigm governs the sequencing of those operations. Existing machines may have sufficient resources to do the work of nodes 3, 4 and 5 simultaneously, even on several points in the matrix, but each firing must involve the temporary storing and fetching of computed data or tokens. For some special purpose applications, particularly in the signal processing area, it is possible to envision building a device which is very much a dataflow computer made of special purpose functional units wired together, but such a device does not support general purpose dataflow computing.

Here, the architectural implications of a dataflow computer are investigated that make use of optical reconfiguration to implement a dataflow graph in hardware. If the computation represented by the graph is repetitive, and if the reconfiguration can be completed in a time which is very short relative to the useful computation time of the graph structure, then a high performance machine can be realized. The machine would have the advantages of the tagged dataflow architectures (easy detection of parallelism) combined with the special abilities of optical computing devices (tightly controlled timing and high speeds). These advantages have been hoped for conventional electronic implementations, but current electronic technology limits how well these advantages can be realized:

> "Apparent disadvantages of instructional-level data flow computers are summarized below:
>
> 1. The data driven at instruction level [sic] causes excessive pipeline overhead per instruction, which may destroy the benefits of parallelism. The long pipeline filling problem is attributed to queuing all enabled instructions at the input ports of every subsystem in the data flow ring. The queue lengths absorb some of the parallelism in a program, thus, performance becomes weak for improper buffering and traffic congestion.
>
> 2. Dataflow programs tend to waste memory space for the increased code length due to the single assignment rule and the excessive copying of data arrays. The damaging effects of the memory access conflict problem are so far not well addressed by dataflow researchers.
>
> 3. When a dataflow computer becomes large with high numbers of instruction cells and processing elements, the packet-switched network used becomes cost-prohibitive and a bottleneck to the entire system.
>
> 4. Some critics feel that dataflow has a good deal of potential in small-scale or very large-scale parallel computer systems, with a raised level of control. For medium-

scale parallel systems, data flow competes less favorably with the existing pipeline, array, and multiprocessor computers."
- [Ref. 30]

In summary, although the dataflow graph captures the form of the computation and inherent parallelism, the maximum parallelism can only be realized with a direct implementation of the dataflow graph. Electronic approaches strive for generality so that performance is sacrificed. With a reconfigurable approach, however, there is no need to create a general fixed architecture since the architecture can be modified to match the form of the dataflow graph.

5.4 Scalable Gaussian elimination processor

As discussed earlier, a potential opportunity for the runtime selected approach is the application of Gaussian elimination to the solution of linear equations. A method is described in Section 3.2 for dealing with pivoting by modifying the address decoder of the coefficient RAM. Here, an alternative approach is considered. The major problem with pivoting is that fast Gaussian elimination implementations make extensive use of pipelining, and the pipelines must be flushed whenever a pivoting decision is made. An improvement is proposed here in which the cost of pipeline flushing is reduced by performing a low precision operation while the main high precision operation is taking place, so that an estimate of pivoting requirements is known prior to completion of the main operation. This appears to be a reasonable approach for a conventional electronic implementation also, however, a large amount of data must be shuttled between the high precision operation and the low precision operation, which is difficult to do effectively with conventional Manhattan style two-dimensional wiring. An optical approach that does not use reconfiguration might solve the bandwidth problem without resorting to a reconfigurable interconnection technology, but then the low-precision logic is underutilized for a number of cycles immediately after its results are known due to the nature of the computation (see next paragraph). The runtime selected model allows the low precision logic to be returned to a logic pool to be reallocated to the high precision computation. This turns out to be appropriate for Gaussian elimination since the bulk of computation occurs immediately after pivoting, and reduces as computation proceeds, allowing a new low precision operation to begin.

A schedule for a 16-stage Gaussian elimination pipeline [31] is shown in Figure 31. The schedule was created by Murdocca for a variation of the OptiComp acoustooptic based processor [32] but the principle of operation applies here as well. The dark banded areas indicate wait states where the pipeline is flushed while it waits for certain parts of the computation to complete. The concept proposed here is to fill the wait states with a low precision calculation of future pivoting needs. The result of the low precision pivoting decision is then used to modify the address decoder mask as described earlier.

TIS - RELEASED FOR DISTRIBUTION

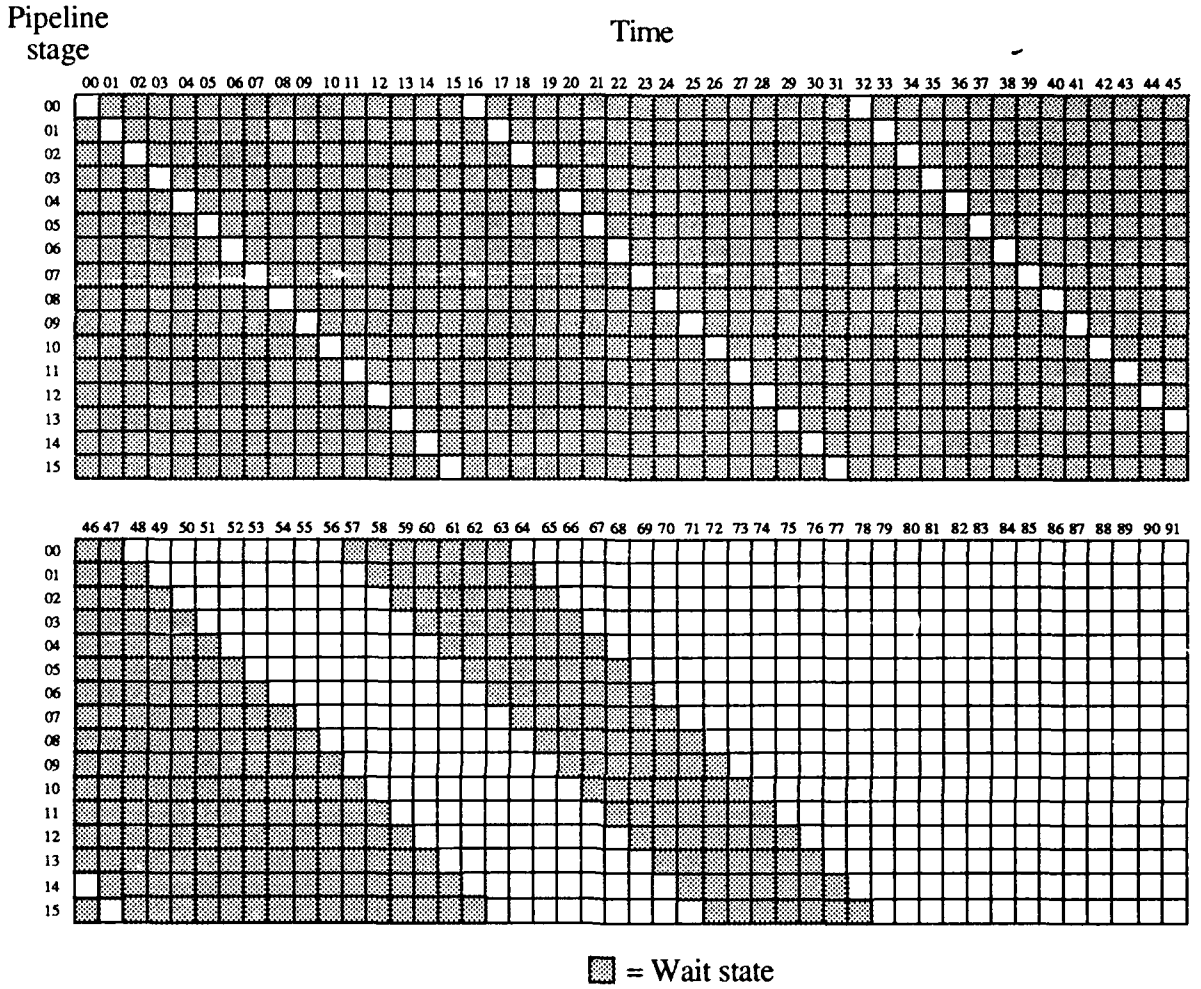


Figure 31: *Schedule of operations for solution of a 10x11 augmented matrix using Gaussian elimination in a 16-stage pipeline. The shaded section of the first 48 stages represent wait states for the first reciprocal operation. The shaded band that begins on time steps 57-63 represent wait states for the first set of multiplications. The wait states are considered for use in a low precision solution.*

6. Putting it all together

A general approach to identifying a suitable reconfiguration strategy for a particular application is to find the approach that yields the greatest performance at the lowest cost. Figure 32 shows a mapping that relates the reconfiguration speeds of various reconfiguration mechanisms with their anticipated influence on computing. The vertical axis indicates speed of reconfiguration with respect to the bit rate of the logic devices. The horizontal axis lists reconfiguration mechanisms discussed earlier in this report from left to right in the order of decreasing expense, where expense

is a qualitative measure of the difficulty in implementing the mechanism. The shading pattern for each entry indicates the relative influence of each reconfiguration mechanism on computing, where heavily shaded regions indicate greater influence than lightly shaded regions. Note that the classifications are based on qualitative evidence discussed in earlier sections, so that the table is open to interpretation. A danger in condensing the results of the Phase I effort in this way is that it may give the impression that there is a single best approach that stands out above the others when in fact, there is a range of approaches that should be matched with the needs of the applications. The main purpose in summarizing the work in this way is to emphasize some distinctions, such as that reconfiguration speeds below the bit rate can be effective in improving performance as for the pivoting problem discussed in Section 3.2.

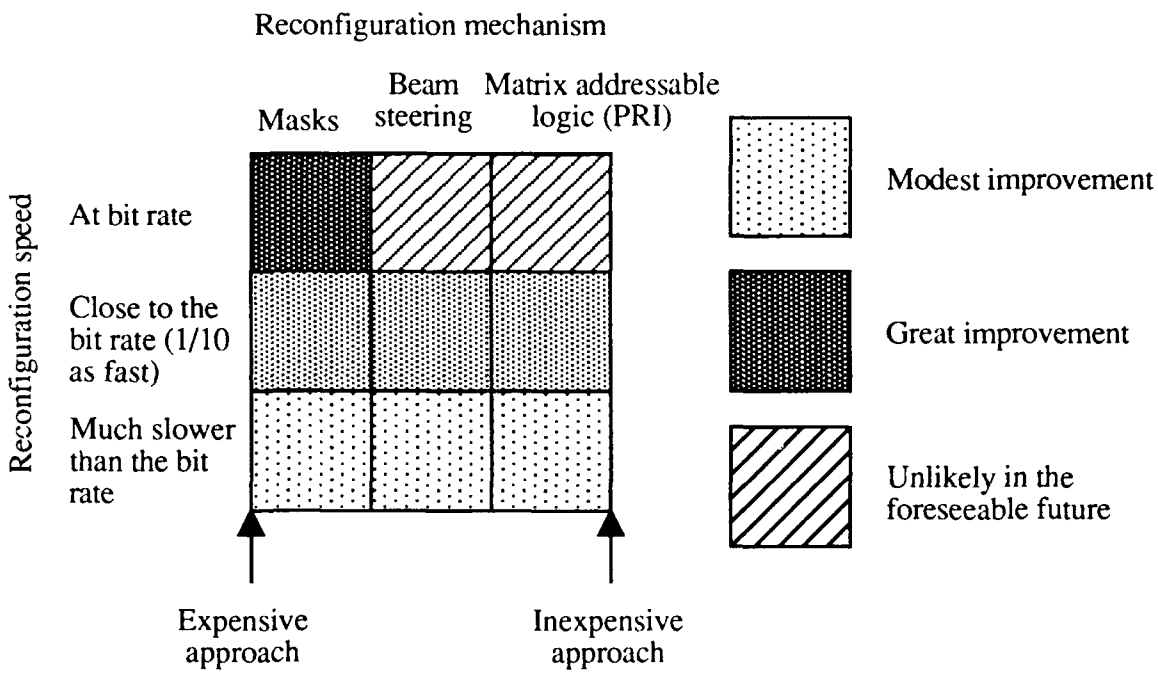


Figure 32: *A qualitative assessment is shown for the reconfiguration speed of various reconfiguration mechanisms. The relative influence on computing for each approach is indicated by the shading pattern.*

The bit rate reconfigurable mask approach is considered to offer the greatest improvement to computing because it can be applied to general purpose computing, which affects nearly every facet of computer usage. However, the slowly reconfigurable beam-steering and matrix addressable (see Section 3 2) logic approaches may be of greater interest to SDIO and the Air Force since matrix operations such as Gaussian elimination and signal processing are important for radar and communication applications. The beam-steering approaches offer greater flexibility in reconfiguration than the matrix addressable logic approach, but matrix addressable logic is more

compact since the reconfiguration mechanism is monolithically integrated. Thus, although the figure favors bit rate masks as having the greatest influence on computing in general, the matrix addressable logic approach may have the greatest impact on SDIO and Air Force needs.

7. Plans for continuing to a Phase II SBIR effort

Current plans for continuing to a Phase II effort include two major thrusts: (1) the development of software development tools, such as a hardware compiler that generates object code for a reconfigurable processor, and (2) the demonstration of a reconfiguration technology, making use of SEED or microlaser based devices.

7.1 Software development tools

The purpose of a conventional high-level language compiler is to translate a high level language such as C or Pascal into the instruction set of a target computer. The purpose of a conventional hardware compiler is to translate high level descriptions of computer architectures into a design, typically at the module-level, the gate-level, or the transistor level. A compiler for a reconfigurable approach combines the high-level language and hardware compilers into one compiler that generates object code for an architecture that it also creates. Such a compiler will be instrumental in generating a more quantitative assessment of the reconfiguration approach (see next paragraph).

Figure 32 provides a qualitative assessment of reconfiguration mechanisms. The organization of the graph is based on subjective evidence, but a more quantitative assessment is preferred. A quantitative analysis of how instruction usage is distributed is proposed for a Phase II effort in order to refine Figure 32 with hard evidence. The quantitative analysis will be steered to investigate hardware caching which was briefly mentioned in Section 3.3, since the influence that hardware caching may have on computing is currently unknown.

7.2 A hardware prototype

A demonstration of a reconfigurable interconnection technology is considered for a Phase II follow on effort. The demonstration would consist of a reconfiguration mechanism such as a reprogrammable mask realized with S-SEEDs or the proposed PRI matrix addressable devices, and input and output channels that mate with optical logic devices. The project will be coordinated with the RL project to avoid duplication of effort and to ensure the practicality of the approach. Dr. Nicholas Craft at AT&T Bell Labs in Holmdel, New Jersey, has offered his expertise for a Phase II effort as a TIS employee. Craft was one of the original optomechanical designers on the Bell Labs all-optical S-SEED based processor reported over two years ago.

8. Collaborations and coordination of efforts

TIS is involved in a number of collaborations and efforts that support the current Phase I effort.

8.1 Interactions with Rutgers University and Rome Laboratory

An optical computing project in the Computer Science Department at Rutgers University in New Brunswick, New Jersey is headed by Murdocca, who serves as the principal investigator on this Phase I SBIR effort. The Rutgers project is involved with developing computer-aided design (CAD) software for the design of digital optical circuits that support the model shown in Figure 2. For example, the circuit layout shown in Figure 9 was generated by the Rutgers CAD tools. The Rutgers group is also involved in the design and construction of an all-optical digital processor in the Photonics Center at Rome Laboratory/Griffiss AFB that is based on the AT&T Bell Labs S-SEED processor that was reported in an AT&T press release two years ago. Dr. Richard Michalak (RL) has offered the use of RL resources to TIS in carrying out the proposed Phase II follow on to the current effort, as well as for other TIS projects (see Section 8.4). Murdocca's interaction with Rome Laboratory is being utilized in this Phase I effort to ensure the practicality of the approach.

8.2 Interaction with Photonics Research Incorporated

Dr. Gregg Olbright and Dr. Jack Jewell of Photonics Research Incorporated (PRI) are working toward the development of matrix addressable logic arrays which will be considered for use in a Phase II followon to this Phase I effort. The basic configuration of the PRI device array is shown in Figure 33. Devices that are located at the crosspoints of active row and column bonding pads are enabled. The advantage of this configuration is that for an N^2 increase in the size of an array, the bonding pad complexity increases by only 2N. A disadvantage is that the computer designer loses a degree of freedom in selecting combinations of logic gates to enable or disable. For example, in Figure 33, there is no combination of active rows and columns that will generate a checkerboard pattern. Despite the limited number of possible on/off combinations for a matrix addressable array, the available complexity is probably sufficient for general purpose computing. Evidence that this may be the case is provided by Murdocca's previous work on using regular interconnects for optical logic circuits [33] in which interconnection topologies are severely restricted when compared with an electronic approach. Although design flexibility is sacrificed, the complexity of the optics and of the electronic addressing is simplified, which are currently more important considerations.

8.3 Interaction with AT&T Bell Labs

Murdocca continues his collaboration with Alan Huang's group in Holmdel, New Jersey, and is involved in organizing a set of CAD tools to support an AT&T sponsored custom SEED and microlaser shuttle. This collaboration resulted from a recent DARPA sponsored workshop on exploring the possibility of a joint academia/industry optical computing consortium.

Murdocca recently visited Scott Hinton's Photonic Switching group in Naperville, Illinois in a coordinated effort with members of the Photonics Center at Rome Laboratory. Hinton's group offered their binary phase grating spot array generation technology to Rutgers and RL as a result of the visit, which will also be made available to TIS. Murdocca is entering into a contractual

relationship with Bell Labs through TIS for organizing an optical computing course to be offered at Bell Labs, which allows Murdocca continued access to Huang's group and Hinton's group.

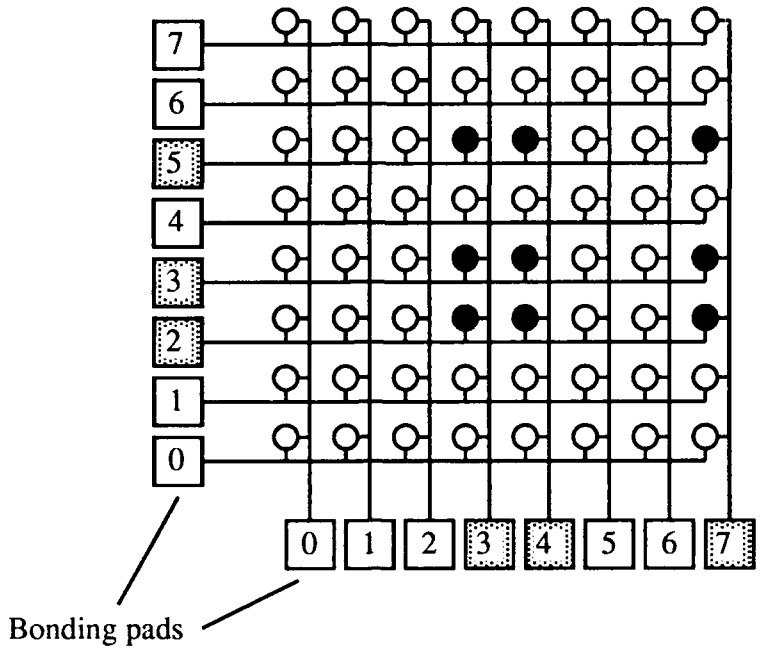


Figure 33: *A matrix addressable array of devices requires only a linear growth in the number of bonding pads for a power of two growth in the number of devices. The indicated pattern is selected by enabling rows 2, 3, and 5 and enabling columns 3, 4, and 7.*

8.4 Coordination with existing TIS Phase II SBIR effort

A Phase II SBIR award was made to TIS through Rome Laboratory. The goal of the Phase II effort is to create interconnection technology for optical logic gates using diffractive elements, and to apply the technology to the RL S-SEED processor. The results from the RL Phase II effort will influence the interconnection technology used for the proposed Phase II follow on to this Phase I effort.

MJM

Miles Murdocca
(Corresponding author)

9. References

[1] Hillis, W. D., *The Connection Machine*, The MIT Press, (1985).

[2] Stark, L., "Microwave theory of phased-array antennas - a review," *Proc. IEEE*, **62**, pp. 1661-1703, (Dec. 1974).

[3] Mailloux, R. J., "Phased array theory and technology," *Proc. IEEE*, **70**, pp. 246-292, (Mar. 1982).

[4] Lentine, A. L., H. S. Hinton, D. A. B. Miller, J. E. Henry, J. E. Cunningham and L. M. F. Chirovsky, "The symmetric self electro-optic effect device," in *Conference on Lasers and Electro-optics*, Technical Digest Series 1987, **14**, (Optical Society of America, Washington, D.C., 1987, 249), postdeadline paper.

[5] Prise, M. E., N. C. Craft, M. M. Downs, R. E. LaMarche, L. A. D'Asaro, L. M. F. Chirovsky, and M. J. Murdocca, "An optical digital processor using arrays of symmetric self-electrooptic effect devices," *Applied Optics*, **30**, no. 11, (Apr. 10, 1991).

[6] Jahns, J. and M. J. Murdocca, "Crossover networks and their optical implementation," *Appl. Opt.*, **28**, 182, (Aug. 1, 1988).

[7] McAulay, A., *Optical Computer Architectures*, John Wiley & Sons, pp. 86-90, (1991).

[8] Ford, J. E., S. H. Lee, and Y. Fainman, "Application of photorefractive crystals to optical interconnections," *Proc. SPIE*, **1215**, paper 16, (Jan. 1990).

[9] Olbright, G. R., R. P. Bryan, K. L. Lear, T. M. Brennan, G. Poirier, Y. H. Lee, and J. L. Jewell, "Cascadable laser logic devices: discrete integration of phototransistors and surface-emitting laser diodes," *Electron. Lett.*, **27**, pp. 216-, (1991).

[10] Jewell, J. L, A. Scherer, S. L. McCall, Y. H. Lee, S. J. Walker, J. P. Harbison, and L. T. Florez, "Low threshold electrically-pumped vertical cavity surface-emitting microlasers," *Electron. Lett.*, **25**, 1123-1124, (1989).

[11] Jewell, J. L., S. L. McCall, Y. H. Lee, A. Scherer, A. C. Gossard, and J. H. English, "Optical computing and related microoptic devices," *Appl. Opt.*, **29**, pp. 5050-5053, (Dec. 1, 1990).

[12] Franklin, M. A., D. F. Wann, and W. J. Thomas, "Pin limitations and partitioning of VLSI interconnection networks," *IEEE Trans. Comp.*, **C-31**, pp. 1109-, (Nov. 1982).

[13] Kiamilev, F., S. Esener, R. Paturi, Y. Fainman, P. Mercier, C. Guest, and S. H. Lee, "Programmable opto-electronic multiprocessors and their comparison with symbolic substitution for digital optical computing," *Opt. Eng.*, **28**, pp. 396-409, (1989).

[14] Tanenbaum, A. S., *Structured Computer Organization*, 3rd ed., pp. 190-202, Prentice Hall, (1990).

[15] Przybylski, S. A., *Cache and Memory Hierarchy Design*, Morgan Kaufmann Publishers, (1990).

[16] Ozguz, V. H., S. H. Lee, S. C. Esener, and B. Monsoorian, "Optical interconnects for microelectronics and parallel computing," UCSD ECE Dept., publication status unknown.

[17] Paturi, R., D.-T. Lu, J. E. Ford, S. C. Esener, and S. H. Lee, "Parallel algorithms based on expander graphs for optical computing," *Applied Optics*, **30**, pp. 917-927, (Mar. 10 1991).

[18] Texas Instruments Incorporated, *The TTL Data Book*, 2nd ed., (1976).

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

[20] Hill, F. J. and G. R. Peterson, *Digital Systems: Hardware Organization and Design*, 3rd ed., John Wiley & Sons, (1987).

[21] Stallings, W., *Computer Organization and Architecture*, 2nd ed., MacMillan Publishing, (1990).

[22] T. Yang and A. Gerasoulis, "A fast static scheduling algorithm for DAGs on an unbounded number of processors," *Proceedings of Supercomputing 91*, Albuquerque, New Mexico, (nov. 1991).

[23] A. Gerasoulis, S. Venugopal, and T. Yang, "Clustering Task Graphs for Message Passing Architectures," *Proceedings of the ACM International Conference on Supercomputing*, Amsterdam, pp. 447-456, (1990).

[24] Kuck, D. J., R. Kuhn, D. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 207-218, (Jan. 1981).

[25] Gurd, R., C. Kirkham, and I. Watson, "The Manchester Prototype Dataflow Computer," *Communications of the ACM*, 28(1), pp. 34-52, (Jan. 1985).

[26] Arvind and G. K. Maa, and D. E. Culler, "Parallelism in Dataflow Programs," *Fifteenth Annual International Symposium on Computer architecture, Honolulu, Hawaii,* May 30-June 2, (1988). Supported on DARPA/ONR N00014-89-J-1988.

[27] Hiraki, K., S. Sekiguchi, and T. Shimada, "System Architecture of a Dataflow Supercomputer," Technical report, Computer Systems Division, Electrotechnical Laboratory, 1-14 Umezono, Sakura-mura, Niihari-gun, Ibaraki, 305, Japan, (1987).

[28] Levy, S., "Muiltiprocessing Computing System with Special Instruction Sequencing," U.S. Patent #3470540, (Sep. 30, 1969).

[29] Arvind and R. S. Nikhil, "A Dataflow Approach to General-Purpose Parallel Computing," *Computation Structures Group Memo 302 (*Jul. 1989). Supported on DARPA/ONR N00014-84-K-0099.

[30] Hwang, K. and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, pp. 747-8, (1984).

[31] Murdocca, M. J. and S. Levy, "Design of a Gaussian Elimination Architecture for the DOC II Processor," Proc. of the 1991 O-E/Lase Symposium, (Jul. 1991).

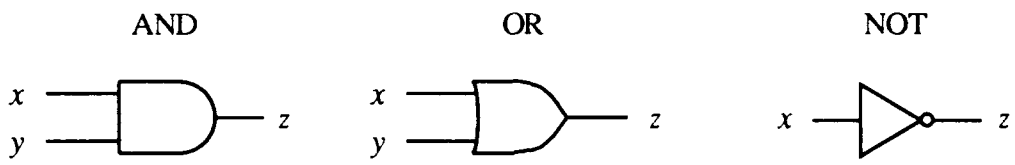
[32] Guilfoyle, P. S., "Digital optical computer fundamentals, implementation, and ultimate limits," *Digital Optical Computing*, Proc. of O-E/Lase '90, SPIE Press, **CR35**, pp. 288-309, (1990).

[33] Murdocca, M., *A Digital Design Methodology for Optical Computing*, The MIT Press, (1990).

Appendix A: Boolean Logic

A review is provided of some basics of Boolean[9] logic.

An AND gate, an OR gate, and a NOT gate can be described by the truth tables shown in Figure A.1. x and y are binary inputs, and z is the output for each case.



x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

x	y	z
0	0	0
0	1	1
1	0	1
1	1	1

x	z
0	1
1	0

Figure A.1: *Truth tables and logic symbols are shown for AND, OR, and NOT Boolean logic gates.*

Truth tables enumerate all possible input combinations and assign a corresponding output for each input combination. The AND function is true (produces a 1) only when x and y are 1, whereas the OR function is true when either x or y is a 1, or both x and y are 1. The NOT gate (also referred to as an inverter) produces a 1 at its output for a 0 at its input, and produces a 0 at its output for a 1 at its input. The inverted signal z is referred to as the complement of x.

Now suppose that a more complex function than a simple logic gate is to be created, such as the majority function shown in Figure A.2. The majority function is true whenever more than half of its inputs are true. This is a common operation used in fault recovery (see Section 5.1).

Since neither AND, OR, nor NOT will implement the majority function directly, the truth table is transformed into a two-level AND-OR equation, and then the majority function is implemented with a suitable arrangement of AND and OR gates. The Boolean equation that describes the majority function is true whenever M is true in the truth table. Thus, M is true when A=0, B=1, and C=1, OR when A=1, B=0, and C=1, and so on for the remaining two cases. The Boolean logic equation is written as shown below, where a bar over a variable means that the complement of the variable is used:

$$M = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

[9]George Boole created Boolean algebra in 1854.

The juxtaposition of two or more variables such as ABC denotes a logical AND operation among the variables. The plus signs '+' in the equation denote logical ORs, and do not imply arithmetic addition.

Majority

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Figure A.2: *A truth table is shown for the majority function. The output* M *is true whenever more than half of the inputs* A, B, *and* C *are true.*

Now that the equation is created, four three-input AND gates can be used to implement each of the terms $\overline{A}BC$, $A\overline{B}C$, $AB\overline{C}$, and ABC, and then the outputs of these four AND gates can be connected to the input of a four-input OR gate as shown in Figure A.3.

In theory, any binary function can be computed in two levels of logic gates given an arbitrarily large stage of AND gates followed by an arbitrarily large stage of OR gates, both having arbitrarily large fan-in and fan-out. For example, an entire computer program can be compiled in just two steps if it is presented in parallel to a Boolean circuit that has an AND stage followed by an OR stage which are designed to implement this function. Although fan-outs of larger than about 10 are usually too costly to implement in many logic families due to the sacrifice in performance, Boolean algebra for two level expressions is still used to describe digital circuits, and then the two-level Boolean expressions are transformed into multi-level expressions that agree with the fan-in and fan-out limitations of the technology. Optimal fan-in and fan-out are argued to be e = 2.7 [A1] in terms of transistor stepping size for bringing a signal off of an electronic chip[10] but the derivation

[10]The result is for driving large capacitive loads and is based on using a sequence of N stages each of which has the same ratio of output to input capacitance, f. For minimum delay this scheme requires $f = e = 2.7$. The result comes from a standard optimization of the function $k \cdot f/\ln(f)$. In practice, any ratio between 2 and 3 does well. This strategy is used for output bonding pads because of the large load capacitance. - *Professor Donald Smith, Rutgers University.*

of that result is based on capacitance of bonding pads and cannot be applied to all aspects of computing since it does not take overall performance into account, which may create local variations that violate the *e* rule dramatically. Electronic digital circuits typically use fan-ins and fan-outs of between 2 and 10.

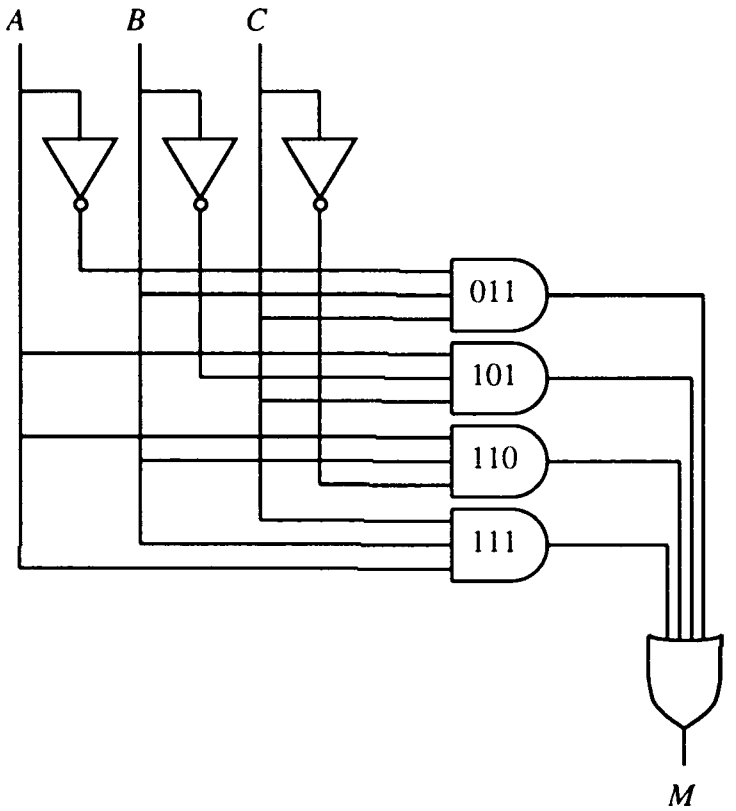


Figure A.3: *A two-level AND-OR circuit implements the majority function. Inverters at the inputs are not included in the two-level count.*

Appendix A references

[A1] Mead, C. and L. Conway, *Introduction to VLSI Systems*, pp. 12-14, Addison Wesley, (1980).

Appendix B: Equivalence of AND-OR and OR-NOR logic

Although AND and OR are traditional logic primitives that are used in representing Boolean equations, the AND function causes difficulties in some technologies. A difficulty with AND is that the logic gate must distinguish between the state in which N-1 of its N inputs are true and the state in which N of its inputs are true. This places a greater burden on the technology than for OR logic where a distinction only needs to be made between the presence or absence of a true input. For this reason, the form of some of the Boolean equations used in this report is the computationally equivalent OR-NOR form. For example, the AND-OR form of the majority function described by the truth table shown in Figure A.2 can be transformed into OR-NOR form as shown below:

$$M = \overline{\overline{\overline{A} + B + C} + \overline{A + \overline{B} + C} + \overline{A + B + \overline{C}} + \overline{A + B + C}}$$

A more subtle problem that is avoided by using OR-NOR logic instead of AND-OR logic is that unused AND inputs do not need to be tied high. For example, if a four-input AND gate is all that is available where a three-input AND gate is needed, then the unused fourth input should be tied to a logical 1 in order for the AND gate to function as intended. This is trivial to accomplish in an electronic technology but can introduce significant complexity for an optical approach since imaging of an additional light source is needed onto the logic gate. For OR-NOR logic, unused inputs typically assume a value of 0, which produces the desired behavior.

Appendix C: Gaussian elimination

A number of engineering problems such as null steering for phased array radar [2.3] require the solution of systems of dense linear equations. Gaussian elimination is one method of solving a linear system that converts an augmented matrix of n rows by n+1 columns (t nxn matrix of coefficients, plus a column which consists of the values for each of the equations) into an upper diagonal matrix, and applies back substitution to solve for the unknowns. For the method used here [C1], a solution is carried out for the system:

$$2x_1 + x_2 + x_3 = 8$$
$$3x_1 - 2x_2 - x_3 = 1$$
$$4x_1 - 7x_2 + 3x_3 = 10$$

The process starts by locating the leftmost column that does not consist entirely of zeros:

$$\begin{bmatrix} 2 & 1 & 1 & 8 \\ 3 & -2 & -1 & 1 \\ 4 & -7 & 3 & 10 \end{bmatrix}$$

Leftmost nonzero column

Then, if the top left element is zero or close to zero, then interchange the top row with another row so that the top left element becomes sufficiently large to take the reciprocal in the next step. This is referred to as the *pivoting* problem, and there are a number of methods for choosing the best pivot element.

Next, compute the reciprocal $1/a$ of the top left element a and multiply the first row by that value:

$$\begin{array}{r} a = 2 \\ 1/a = .5 \end{array} \begin{bmatrix} 1 & 1/2 & 1/2 & 4 \\ 3 & -2 & -1 & 1 \\ 4 & -7 & 3 & 10 \end{bmatrix}$$

Then add suitable multiples of the top row to the remaining rows so that only the top row has a leading nonzero element:

$$\begin{bmatrix} 1 & 1/2 & 1/2 & 4 \\ 0 & -7/2 & -5/2 & -11 \\ 0 & -9 & 1 & -6 \end{bmatrix} \quad \begin{bmatrix} 1 & 1/2 & 1/2 & 4 \\ 0 & 1 & 5/7 & 22/7 \\ 0 & -9 & 1 & -6 \end{bmatrix}$$

Repeat on the next smaller submatrix until the matrix is in upper diagonal form:

$$\begin{bmatrix} 1 & 1/2 & 1/2 & 4 \\ 0 & 1 & 5/7 & 22/7 \\ 0 & 0 & 52/7 & 156/7 \end{bmatrix}$$

Thus we have: $x_1 + 1/2x_2 + 1/2x_3 = 4$

$x_2 + 5/7x_3 = 22/7$

and $52/7x_3 = 156/7$.

Applying back substitution, solving for x_3 first yields $x_1 = 2$, $x_2 = 1$, and $x_3 = 3$.

Appendix C references

[C1] Anton, H., *Elementary Linear Algebra*, 2/e, John Wiley & Sons, pp. 8-16, (1977).