NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents

The Design of PAAP:
Programmable Asynchronous Array Processor

Marco A. Zelada

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master in Computer Science at
Concordia University
Montréal, Québec, Canada

April 1992

ISBN 0-315-73669-0

ABSTRACT

The Design of PAAP: Programmable Asynchronous Array Processor

Marco Zelada

This thesis explores architectural features appropriate to massively parallel architectures such as, Processing Element (PE) and Routing Element (RE) programmability within an array processor environment, unrestricted data flow direction, and true array scalability. We illustrate these features through an array design, which we term the Programmable Asynchronous Array Processor (PAAP).

The PAAP proposes a methodology for mapping high-level computation into hardware structures. This feature is shared with systolic and wavefront arrays. However, the PAAP differs in that its PEs can be dynamically programmed to work in an MIMD fashion. They can also be interconnected via dyadic programmable REs to form asynchronous pipelines. In contrast, most systolic and wavefront arrays work in an SIMD fashion, have static interconnections and implement a single special purpose hard-wired instruction. Those which are programmable and work in an MIMD fashion lack the interconnection reconfigurability and data flow control present in the PAAP.

PE and RE instructions and data are not fetched from memory. Instead, the PE and RE are programmed to execute the desired instruction and routing scheme during the program load phase, and data flows through the complete array during the program execution phase. The essence of the architecture is captured by the configurable routing which gives the PAAP its flexibility and power. The RE is a four way bidirectional router which provides maximum flexibility in the definition of pipeline routes. It permits a pipeline to be defined in any direction across the array, downward, sideways in either direction, and upwards.

The PAAP addresses the three main asynchronous circuit problems, namely, computational interference, signal ordering, and transfer interference. It uses a modified two-phase signal protocol which allows the active and passive end of the circuit to implement their own return-to-zero synchronization upon the receipt of the proper protocol sequence [BrozozowskiJ89].

Acknowledgment

I would like to thank some of the people that made this work possible through their patience and understanding. First, I want to thank Dr. T. Fancott for his faith in me and his excellent guidance, which helped me carry this project to completion. Thanks also go to Henry Polley, with whom I spent many long hours discussing the architecture and design issues, and without which the thesis text revisions would not have possible.

My deepest thanks go to my wife Li-Yuin Tam for her devoted moral support.

Thanks also go to Hala Tabl and Sam Alexander for their invaluable review comments. The list of other people that have influenced my research and helped in one way or another is too long to mention here. I would like to just say thank you all.

Table of Contents

Acknowledgment ii
Table of Contents iii
List of Tables iv
List of Figures v
Chapter 1 Introduction 1
Section 1.1 The Motivation of This Research 2
Section 1.2 Thesis Introduction 4
Chapter 2 Parallel Architecture Concepts 5
Section 2.1 Parallel Processing 5
Section 2.2 Parallel Architecture Taxonomies 6
Section 2.3 Architectural Issues 9
Section 2.4 Parallel Processing Implementations 11
Section 2.5 MIMD Parallel Architecture Overview 14
Section 2.6 Systolic and Wavefront Arrays 17
Section 2.7 Asynchronous Design 19
Section 2.8 The CHiP Architecture 21
Chapter 3 PAAP Architecture Overview 22
Section 3.1 Introduction to the PAAP Array 22
Section 3.2 PAAP Array Building Blocks 24
Section 3.3 The Smallest PAAP Array Configuration 26
Chapter 4 The PAAP Programming Paradigm 38
Section 4.1 Operational View of the Architecture 38
Section 4.2 PAAP Programming Variants 41
Section 4.3 Examples of Algorithmic Units 42
Section 4.3.1 Pre-increment 42
Section 4.3.2 Swap 43
Section 4.3.3 Post-decrement 44
Section 4.3.4 Arithmetic and Logic Expression Evaluation 44
Section 4.3.5 Inclusion of Constants in the Computation 45
Section 4.3.6 Boolean Expression Evaluation 46
Section 4.3.7 High Level Language IF Construct 47
Section 4.3.8 High Level Language Looping Constructs 48
Section 4.4 Examples of Complex Operations 49
Section 4.4.1 Min/Max Search Algorithmic Units 50
Section 4.4.2 A Sort Algorithmic Unit 52
Section 4.4.3 A Match Algorithmic Unit 54
Section 4.4.4 A Factorial Algorithmic Unit 55
Chapter 5 The PAAP PE and RF Architecture 57
Section 5.1 A Processing Element 57
Section 5.2 A Routing Element 73
Chapter 6 Practical Issues 79
Section 6.1 The Design Environment 79
Section 6.2 A Simulation Example 81
Chapter 7 Conclusion 85
References 87
Appendix A Programming the PAAP 91
Appendix B Automatic Vector Generation Setup Files 96
Appendix C PAAP Detailed Circuit RNL Netlist 108
Appendix D Automatic Simulation Makefile 125
Appendix E Automatic Simulation Step Output 126
Appendix F Partial Simulation Results 127

List of Tables

TABLE 1. Feature Size Shrink Trend 3
TABLE 2. Parallel vs. Sequential Computation 3
TABLE 3. Addressable PAAP Registers 29
TABLE 4. Byte Parallel Operations 32
TABLE 5. The PAAP Array Control Register 33
TABLE 6. The PAAP Array Status Register 35
TABLE 7. PE Instruction Set 39
TABLE 8. PE Output Routing 40
TABLE 9. RE Instruction Set 41
TABLE 10. The ALU Status Bus Bits 67

List of Figures

FIGURE 1. Architecture and IC Performance Growth 1
FIGURE 2. Design Rule Shrink Trend 2
FIGURE 3. A PAAP Array 23
FIGURE 4. PAAP Building Block ICs 25
FIGURE 5. A 1089 PE PAAP Array Configuration 25
FIGURE 6. The Smallest PAAP Array Configuration 26
FIGURE 7. A Possible Host/Array Interface 31
FIGURE 8. A Pre-increment Implementation 43
FIGURE 9. A Swap Implementation 43
FIGURE 10. A Post-decrement 44
FIGURE 11. An Arithmetic and Logic Expression Evaluation Implementation ... 45
FIGURE 11. An Inclusion of Constants Implementation 46
FIGURE 13. A Boolean Expression Evaluation Implementation 46
FIGURE 14. An IF Statement Implementation 47
FIGURE 15. A FOR Loop implementation 48
FIGURE 16. Deadlock Synchronization 49
FIGURE 17. A Max Algorithmic Unit Implementation 50
FIGURE 18. A Min Algorithm Implementation 51
FIGURE 19. A Max Algorithm Implementation 51
FIGURE 20. A Descending Sort Algorithmic Unit Implementation 52
FIGURE 21. An Ascending Sort Algorithm Implementation 53
FIGURE 22. A Match Algorithmic Unit 54
FIGURE 23. A Match Algorithm Implementation 54
FIGURE 24. A Partial Factorial Program 55
FIGURE 25. The Processing Element Block Diagram 57
FIGURE 26. Inter-PE Synchronization 59
FIGURE 27. Input Synchronization Module Schematic Diagram 62
FIGURE 28. Input Synchronization Module Timing Diagram 63
FIGURE 29. Output Synchronization Schematic Diagram 65
FIGURE 30. Output Synchronization Timing Diagram 66
FIGURE 31. PE Core Schematic Diagram 68
FIGURE 32. PE Core Instruction Loading and Execution Timing Diagram 69
FIGURE 33. Control Unit Schematic Diagram 71
FIGURE 34. ALU Schematic Diagram 72
FIGURE 35. RE Schematic Diagram 76
FIGURE 36. RE Instruction Loading and Execution Timing Diagram 77
FIGURE 37. Bit-route Schematic Diagram 78
FIGURE 38. The Oct Tool Set Flow Chart 80

Chapter 1 Introduction

The speed and power of computers is growing at a rate unimaginable a decade ago. Many factors have contributed to this growth, such as improved IC production techniques and architectural innovations. The interaction between computer architecture and IC technology is complex and bidirectional. The characteristics of various IC implementations affect decisions architects make by influencing performance, cost and other attributes. At the same time computer architecture developments impact the viability of IC technologies by emphasizing different technology characteristics such as density, power and speed (see figure 1) [HennessyJ91].

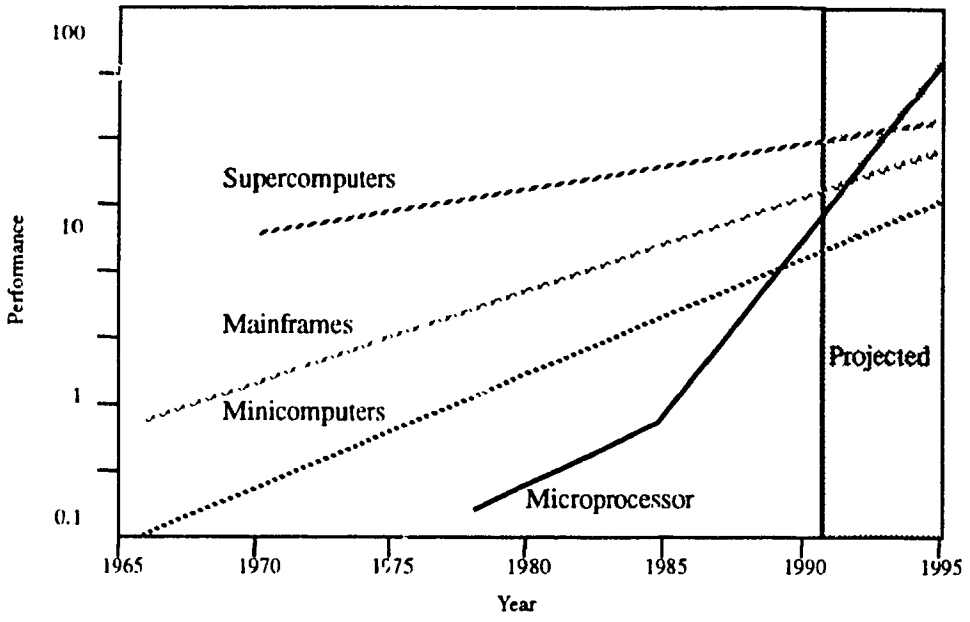


FIGURE 1. Architecture and IC Performance Growth [1]

This growth is also fueled by increased requirements placed on the hardware by new software packages and by intense commercial competition, which forces IC manufacturers to continuously improve their products. The coming of age of standards in operating systems, communications, and software interfaces has thrown the market open to a broad spectrum of manufacturers who can concentrate on specific aspects of the industry without having to cope with all facets involved in producing an integrated hardware/software system. Examples of such industry specialized products are: microprocessors (Intel, Motorola), computers (IBM, Apple), CAD tools (Cadence, Mentor), etc. Nevertheless, such high growth rate has brought the industry close to the physical barriers inherent in any given semiconductor material, such as minimum feature size and maximum speed [RettbergR90][PeaseD91].

---

1 Source: Hennessy J. L. and Jouppi N. P., "Computer Technology and Architecture: An Evolving Interaction", IEEE Computer Vol 24, No 9, pp 18-28, 1991

1.1 The Motivation of This Research

As demands emerge for greater computer processing speeds and capabilities, traditional sequential processors have begun to reach their performance upper bounds. Advanced IC fabrication technology improvements have accelerated this computer revolution by constantly shrinking device sizes (see figure 2) [VoltmerF85].

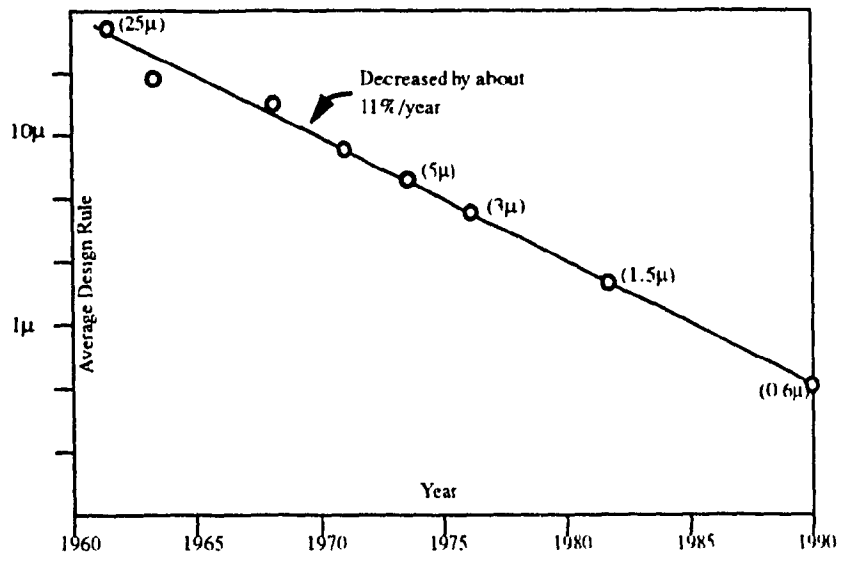


FIGURE 2. Design Rule Shrink Trend [2]

The major benefit of scaling down feature sizes is the ability to increase the number of devices that can be integrated onto a single IC without changing the power supply requirements. This has lead to exponential improvements in IC complexity, which translates into increased performance and cost per function. The average die size has steadily increased despite the impact of periodic feature size reductions, (see table 1). The feature size prediction for the end of this century in table 1 is based on the current research patterns and published papers from places like IBM and AT&T Research Labs. The literature reports experimental devices made under very controlled conditions where they measure the feature size of working devices in terms of the number of aligned molecules [WeberS88]. However, this improvement trend cannot continue for much longer because current IC processing technologies are approaching the physical limits below which silicon products become unreliable. The current generation of processors are being manufactured at around 0.8 micron feature sizes, and feature sizes of usable commercial devices below 0.1 microns are expected before the end of the century [SantoB88] [SavariaY(a)86] [SavariaY(b)86].

---

2 Source Voltmer F W., and Jones N W., "Factors Contributing to Increased VLSI Density", VLSI Handbook, Chapter 1, Figure 3, Florida Academic Press, p 4, 1985.

TABLE 1. Feature Size Shrink Trend [3]

Decade	*Chip Size*	*Feature Size*	*Transistors Per IC*
1960	1 x 1 mm	30.0 Microns	20
1970	2.5 x 2.5 mm	10.0 Microns	1250
1980	6 x 6 mm	3.0 Microns	80,000
1990	12 x 12 mm	0.8 Microns	1,500,000
2000	25 x 25 mm	0.1 Microns	10,000,000

Device feature size reduction alone will not guarantee continued performance improvements and thus other avenues must be explored. Parallel processing is the obvious choice for continued performance improvements beyond the current sequential architectures and physical limits of the known semiconductor materials.

A typical example of the order of magnitude improved performance attainable by parallelizing sequential algorithms can be found in general problem-solving techniques such as state-space search, which are widely used in AI and VLSI CAD applications. Its sequential implementation is computationally intensive; yet, it experiences an almost linear speedup as the number of parallel processors increases. This performance increase trend continues even with a very large number of parallel processors. The literature shows that one of the best ways to improve performance is to adopt parallel processing techniques over their equivalent sequential ones (see table 2) [GottliebA82] [KumarV90] [KumarV87] [NagashwaraV87] [ArvindamS90].

TABLE 2. Parallel vs. Sequential Computation [4,5]

Algorithm	*Parallel Time*	*Sequential Time*
Permutations of N objects	O(logN)	O(N)
Partial sums of N elements	O(logN)	O(N)
Merging 2 lists of N elements	O(logN)	O(N)
Sorting N elements	$O(\log^2 N)$	O(NlogN)
FFT	O(logN)	O(NlogN)

3 Source: Forester T., "The Microelectronics Revolution: The Complete Guide to the New Technology and Its Impact on Society", Cambridge MA: The MIT Press, 1985

4 Source: Gottlieb A. and Schwartz J.T., "Networks and Algorithms for Very-Large-Scale Parallel Computation", IEEE, Computer, Vol. 15, No. 1, pp. 27-34, Jan 1982

5 This table assumes that there is an infinite number of processors to implement the algorithm.

1.2 Thesis Introduction

Architectural innovation is the path to pursue in order to ensure rapid and continued growth. In the short term the easiest and most accessible route is a superscalar architecture which analyses instruction streams and allocates processing units according to a special algorithm which ensures maximum instruction overlap. Alternatively, multiple computer architectures can be used to distribute the task to particular server units; however, architectures must ultimately move to massive parallelism to afford continued growth and computing power [Suaya90].

This thesis explores some new architectural features appropriate to massively parallel architectures, and illustrates these features through a model processor array design, which we term the Programmable Asynchronous Array Processor (PAAP).

PAAP is but a small element of an overall massively parallel architecture design, and consequently many issues peripheral to the array design remain simply defined, but not illustrated. The full design of a computer is beyond the scope of this thesis; however, array programmability aspects are explored and demostrated by simulation. Relevant circuit elements designed using both university and commercial level tools, such as the Oct Set of Tools from UC Berkeley and the Cadence Set of tools from Cadence Design Systems, are illustrated.

Advances in IC fabrication technology have made it possible for VLSI architectures such as the PAAP array design to be feasible. We can conceive of large numbers of PEs and REs (sub-array units) being built as packaged ICs with current technology. These packaged sub-array units could then be used to build larger arrays on standard extension cards that may be attached as peripheral co-processor cards to standard compute engine hosts.

The thesis is organized into seven chapters. In Chapter 1 we find a general thesis introduction that briefly describes the work and motivation. In Chapter 2 we compiled the relevant theoretical background into a focused set of reference information points that span parallel architectures and design, current architecture implementations, an MIMD overview, systolic and wavefront arrays, asynchronous concepts, and wafer scale integration. Chapter 3 describes the minimum computational environment that the PAAP array requires, as well as its functional operation. Chapter 4 provides a programming example and looks at possible related software issues. Chapter 5 describes the PAAP PE and RE design, schematic level and timing diagrams are included for the most important subcomponents. Chapter 6 discusses practical issues such as the design environment used, and overviews simulation results. Chapter 7 is the conclusion, which summarizes the work and outlines design shortcomings and possible future enhancements.

Chapter 2 Parallel Architecture Concepts

The most relevant parallel processing concepts to this thesis are presented in this chapter. First we look at the original computer architecture taxonomy by Flynn and an updated taxonomy by Duncan. The reader may then be able to properly place the PAAP array design within the existing parallel architecture taxonomies. We then examine the architectural issues faced in designing a massively parallel architecture and examine currently available implementations. The reader should have an appreciation for the design choices taken in the PAAP array design. Last, we overview the main characteristics of MIMD architectures as well as wavefront arrays and asynchronous design techniques. This will give the reader an understanding of the PAAP array as a reconfigurable-programmable-MIMD wavefront array.

2.1 Parallel Processing

The decades of the 80s and early 90s have seen the introduction of a wide variety of commercial parallel architectures that complement and extend the major approaches to parallel processing developed in the 60s and 70s. Their design motivation and market areas cover a broad spectrum: (a) Transaction processing systems, such as parallel UNIX systems for data processing applications (i.e. The Balance from SEQUENT), (b) Numeric supercomputers such as Hypercube systems for scientific and engineering applications (i.e. The iPSC from INTEL), (c) VLSI architectures, such as parallel microcomputers for exploiting very large scales of integration (i.e. The transputer from INMOS), and (d) Neurocomputers, such as the Connectionist computers for general purpose pattern matching applications (i.e. The Connection Machine (CM) from THINKING MACHINES) [TreleavenP88].

Massive parallel processing can be thought of as a problem of event scheduling and resource management. These resources can be memory or processing elements as well as communication channels, and the events refer to computation requests. In order for any real computation to take place, in a parallel system, operations have to be properly scheduled on the available resources. Previously it was believed that processing and switching elements were more important and thus system performance was based on measuring how well such units performed with no regard for how they were interconnected. At the present time it is believed that interconnections are the ultimate performance limit on any type of architecture.

Minimizing the cost of Inter-PE communication and at the same time providing a flexible enough communication scheme is always desirable. Unfortunately, striking the perfect balance between low communication overhead and flexibility is not a simple task. If the algorithm being executed does not map the physical connectivity of the machine well, then we may find an excessive amount of idle processors due to communication overhead. The ultimate communication scheme would adapt its connectivity configuration to the algorithm being performed [KuckD86].

Parallel architectures have problems which have prevented them from being widely accepted. [SakaiS90] makes reference to at least 5 different areas which need great improvements before general-purpose parallel computers can achieve significant higher performance:

- Extracting the maximum parallelism from a user program automatically,
- Minimizing synchronization overhead,
- Minimizing communication overhead,
- Obtaining the optimal load balancing, and
- Ensuring reliability.

Many proposed parallel architectures have addressed some or all of these issues, but there is still much more work ahead because users still perceive parallel computers as being not general purpose. Another problem is that they are difficult to program because either their programming environments still use the sequential model of computation adapted for the target parallel architecture, or one must learn the underlying architecture in great detail before attempting to write any useful programs. Both approaches are clumsy and time consuming, yet, attempts are underway to correct these problems by the development of object-oriented languages and of sophisticated software development environments that automatically extract the parallelism embedded in user programs and optimize such programs for the target architecture.

However, general trends of research are looking forward to bringing general purpose parallel processing and many commercially available parallel computers are beginning to show promising results in their software environment implementations, in the adoption of standards on such crucial system components as the Operating System and User Interface. Other factors influencing the proliferation of parallel processing technology are the advances in VLSI architecture and the emergence of new parallel architectures [DenningP86] [KatevenisM84] [MelvinS87].

2.2 Parallel Architecture Taxonomies

In his classic paper, which still retains its validity, Flynn proposed a taxonomy for parallel computers which defines four classes of Von Neumann-based computers depending on the multiplicity of the instruction and data streams [FlynnM56]:

- Single Instruction Single Data (SISD) corresponds to the sequential Von Neumann machine we all know, which can only execute one instruction and work on one data item at a time.

SISD computers are traditionally Von Neumann uniprocessors, but by a clever extension of the concept of horizontal microprogramming it is possible to connect multiple uniprocessors and control them simultaneously. The approach is called Very Long Instruction Word (VLIW). The idea is that if a 32-bit uniprocessor can perform the required computation, by adding the proper control structures, one may be able to extend the 32-bit to 64-bit or 128-bit word size. Operands are thus properly partitioned to the appropriate section to work on their own section of the word, and at the end, their results are collected and the result word is assembled from the partial word results generated by each of the processors. The main difference between VLIW and MIMD machines is in the very long instructions that control the machine, using words of up to thousands of bits long [FisherJ84].

- Single Instruction Multiple Data (SIMD) corresponds to the parallel execution of a single instruction. This types of architectures include vector-arrays, array processors, the CM, etc.

  SIMD machines are usually composed of a single Control Unit (CU) which fetches and decodes instructions, and some N number of interconnected Processing Elements (PE). Once the instruction is on line to the CU, it is either executed by the CU (a jump instruction) or it is broadcast to the PE pool for execution. The PEs operate synchronously, but each has its own memory which may contain different data items and individual partial results. SIMD architectures can come in different arrangements: vector processors, array processors, associative processors. This classification is done based on the CU's complexity, the processing power and addressing method of each PE, and the interconnection facilities between PEs. In array processors the CU has limited capabilities and its PEs communicate through a connection networ[1]. Array processors are thus well-suited for grid problems and in some cases vector processing. Other implementations make the CU a full-size computer which communicates with PEs via message-passing schemes [HordR(a)90].

- Multiple Instruction Single Data (MISD) corresponds to pipelined processors that allow for consecutive instructions of a program to be in different stages of execution by advancing through pipelines of functional units in a staggered fashion, one function at a time. Examples of this architecture include machines based in high-end processors such as the 80486 or the 68040.

MISD-type architectures utilize consecutive program execution modules. At any given time, a different part of an instruction is executing in stages and it advances through the pipelines of functional units in a staggered fashion. The pipelined processor is an MISD processor which partitions each instruction into simpler computational steps that can be executed independently by functional units. Each one of these computational units is called a pipeline segment.

- Multiple Instruction Multiple Data (MIMD) could be better described as a number of SISD configured to communicate among themselves in the course of a program.

  In MIMD architectures, several processors operate in parallel in an asynchronous manner and share access to a common memory module. Each processor is capable of running its own instruction, which allows for a problem to be subdivided and mapped onto an array of MIMD-type PEs. Data also flows across PEs as initial data and partial or fully computed results. Data can be used to synchronize the machine and some PEs may have to wait for other PEs to generate the proper results [SutherlandJ89].

In 1990, Duncan updated Flynn's taxonomy by adding architectures that can not be easily accommodated within Flynn's taxonomy. For example, pipelined vector processors exhibit substantial concurrent arithmetic execution and can work on hundreds of vectors in parallel. However, they lack the single instruction execution property found in SIMD architectures and do not classify as MIMD because their PEs are not asynchronously autonomous. Duncan's taxonomy also excludes architectures that only provide low-level parallel mechanisms such as instruction pipelining, multiple CPU functional units, and separate CPU and I/O processors. The reasoning behind this exclusion is that even though they contribute to an increase in performance they do not make the architecture parallel. The complete taxonomy can be summarized as follows [DuncanR90]:

- Synchronous: Concurrent operations are coordinated in lockstep via global clocks, central control units, or vector unit controllers.
  - Pipelined vector processors: Characterized by multiple, pipelined functional units, which implement the needed operations for both vector and scalar operands, and which can work concurrently.
  - SIMD: As defined in Flynn's taxonomy.
    - Processor arrays: Implemented as an array of special purpose processors with local connectivity between them and the local memory associated with each processor.

- Associative memory: They use a special comparison logic to access and store data in parallel according to its contents.
- Systolic arrays: Pipelined multiprocessors in which data is pulsed in a rhythmic fashion from memory and through a network of processors before returning to memory.

- MIMD: As defined in Flynn's taxonomy.
  - Distributed memory: The nodes share data by explicitly passing messages via the interconnection network which can be one of the following topologies: ring , mesh, tree, hypercube , reconfigurable.
  - Shared memory: Interprocess coordination is done via a global shared memory that each processor can access via: bus, crossbar , multistage networks.
- MIMD paradigms: These architectures are considered hybrids which are based on MIMD principles such as asynchronous operation and multiple operation and data streams; however, they also exhibit distinctive organizing principles which are attributed to other architecture types.
  - MIMD/SIMD: Selected parts of the MIMD machine can be controlled in an SIMD fashion.
  - Dataflow: Instructions are executed as soon as all of the needed operands are available.
  - Reduction: Instructions are executed when its results are required as operands for another instruction already enabled for execution.
  - Wavefront arrays: They combine the systolic data pipelining concept with an asynchronous dataflow approach.

2.3 Architectural Issues

Parallel architectures can be classified by their granularity because it is the size of work units that are allocated to a single processor. The main subdivisions are coarse and fine grain. Coarse-grain parallel architectures can be distinguished by the low number of complex processors. On the other hand, fine-grain parallel architectures utilize a large number of very simple processors. There are also some researchers that would further divide parallel architectures by grain and define medium-grain architectures as a compromise between coarse and fine grain architectures. Yet in a complex system, the actual delivered performance is algorithmically-specific and not architecturally bound. This is related to how well a particular architecture is suited for the particular problem being processed [MohanJ83].

Fine-grain parallelism has the potential of being faster, provided that the algorithm maps very well into the mesh of processors present in the machine's architecture.

Coarse architectures are better understood because of their similarity to sequential processing. This influences the state of software technology that is available for such parallel systems. It may also explain why it is not uncommon to find parallelizing compilers that retarget languages such as FORTRAN, C, PASCAL, etc., to coarse-grain computers such as the Encore Multimax, and no parallelizing software for fine-grain computers such as the CM [KruatrachueB88].

Even though there is an attempt by some portions of the parallel architecture research community to make their designs commercially acceptable via their ease of programming, most of the research effort has concentrated on speeding up solutions to specific problems or classes of problems. As a result, the large number of proposed parallel processing architectures exhibit such great diversity of concept combinations.

The most important issues to consider while designing a parallel architecture are [BasuA84]:

- Loosely vs. tightly coupled systems: The distinction here is based on how the memory system is structured, whether each processor has access to its own local memory (loosely) or the memory system is a global shared memory model (tightly).

- Parallel vs. distributed: The divergence point here is the way the computation gets done, whether the computing elements are concentrated in one place and work in parallel with very little communication overhead (parallel), or scattered around and have to incur a large communication overhead (distributed).

- Shared memory vs. message passing: This division follows from the memory system structure. Depending on whether the system is tightly or loosely coupled, processor communication can take place via shared-memory variables or message-passing.

- Synchronous vs. asynchronous: The distinction here is based on how the PEs synchronize with each other. Synchronous systems utilize a general system-wide clocking scheme. Asynchronous systems do not have a global clocking scheme, they utilize local clocking schemes that ensure timing error-free computation.

- SIMD vs. MIMD: The main distinction here is the number of instructions that can be executed in parallel by the system.

- Special vs. general purpose: This division is done on the basis of how flexible the PE interconnection network is and the generality of the computation that each PE can perform.

Some proposed designs such as systolic arrays provide static PE interconnections, others provide programmable PE interconnections. Most PEs have some level of programmability, but the issue is how general purpose such basic instructions are.

2.4 Parallel Processing Implementations

The advent of the supercomputer term occurred in the 1975 time frame when it was applied to the CDC-7600, the Illiac-IV and other high performance machines of the day. The term became firmly established with the introduction of the CRAY-1. This section briefly covers a few interesting parallel computers that in most cases classify as today's supercomputers.

We start with the Illiac-IV which was the first large scale array computer that incorporated high levels of parallelism and pipelining. Operational in 1975, it consisted of a single CU that broadcast instructions to its 64 PEs. Each PE has 2K 64-bit words of working memory and the main memory is implemented on disk with a capacity of 8 million words at a transfer rate of up to 500 megabits per second.

The CU has access to all of the core memory with a cycle time of 60ns. The CU fetches and decodes all instructions; after decoding, some instructions are broadcast to the PEs for execution, while some other instructions are executed by the CU. There are three data paths available for communication among PEs and between PEs and the CU.

- The CU can access all of the core memory so it can load or store data on any one of the PE's local memory.
- The CU can communicate with all PEs by broadcasting the same word to all PEs simultaneously.
- The PEs can communicate with each other via the ROUTE instruction which transfers data from a source PE to a destination PE.

The Illiac-IV was capable of performing 300 MIPS in 32-bit mode, and even though it was built using electronic components from the late 60s, for certain important classes of applications it was the fastest computer of its time [HordR(b)90].

A more recent computer, the CM, was first introduced in 1986. The 1000 MIPS CM-1, used up to 64K 4K-bit RAM, single-bit processors. The second connection machine, the 2500 MFLOPS CM-2, was introduced in 1987. It still used up to 64K single-bit processors, but this version provided 8K-byte RAM per processor, faster clock speed, and floating point hardware support.

The CM's interprocessor communication network is called a hypercube. In this scheme there are up to 2^N nodes, and each node has N directly connected nodes; the longest path between any two nodes (diameter) has at most N steps through other nodes; in the CM's diameter is 16. The heart of the system is the parallel processing unit, which consists of thousands of processors, each with thousands of memory bits which can be considered as shared memory. The processors can process data stored in their local memory, as well as form logical interprocessor connections so that data can be exchanged among processors. An important concept is that the CM implements parallel programming constructs directly in hardware. Parallel data structures are spread across the data processors with a single element stored in each processor's memory. When parallel data structures have more than 64K elements, the hardware operates in virtual processor mode, presenting the user with a large number of processors, each with a correspondingly smaller memory. Scalar data is held at the front-end processor, which also controls the overall data parallel program. Program steps that require parallel data are passed over an interface to the CM parallel processing unit, where they are broadcast for execution by all processors at once. Front-end controllers, of which there can be up to 4, provide the programming environment for the CM; they also serve as network communication gateways and for storing programs. For every group of 8K data processors there is one I/O channel. Each channel may be connected to either a high-resolution graphics display framebuffer module or one general I/O controller supporting an I/O bus to which several DataVault mass storage devices may be connected.

Data processors are implemented using four chip types. A proprietary custom chip that contains the ALU which can execute variable length operand instructions, flag bits, router interface, NEWS grid interface, I/O interface for 16 data processors, and proportionate pieces of the router and NEWS grid network controllers. The memory consists of commercial RAM chips. The floating-point accelerator consists of a custom floating-point interface chip and a floating-point execution chip; one is required for every 32 data processors.

The CM-2 parallel unit has a structured communication mechanism called the NEWS grid. In the CM-1 the grid is a fixed two-dimensional grid, but the CM-2 supports programmable grids with arbitrarily many dimensions. The NEWS grid allows processors to pass data according to a regular rectangular pattern. For example, in a two-dimensional grid each processor could receive data from its neighbor to the east thus shifting the contents of the grid one position to the left. The advantage of the NEWS grid over the router is that the routing overhead of specifying destination addresses is eliminated. This is an optional optimization that some applications can use [HordR(c)90].

One of the disadvantages of the CM is that its computation gets done in an SIMD single-bit fashion, as well as the difficulty in scaling the machine beyond 64K processors without a major redesign effort. An

interesting alternative is the 32-bit INMOS transputer, which was designed as a basic building block for processor arrays of any size. It provides a direct implementation of a message-passing loosely coupled architecture.

The transputer architecture was targeted for the efficient execution of programs written in the OCCAM parallel processing language. In OCCAM programs, the basic computational modular component is the process that communicates with other concurrent processes through channels. A transputer contains a processor that in some models includes a floating-point unit, a 4K byte memory, and 4 standard point to point communication links that allow direct connection to other transputers. Concurrency is supported by hardware support for scheduling. The design philosophy is in accordance with the RISC approach; however, repetitive operations such as multiply and block moves are implemented in microcode with hardware assistance [DeCegamaA(a)89].

Another interesting implementation is the WARP machine from CMU, which is a high-performance systolic array computer designed for computation-intensive applications. There are three major components to the system: the WARP processor array, the interface unit (IU), and the host.

The WARP array performs the computation-intensive routines such as low-level vision routines or matrix operations. The IU handles the I/O between the array and the host and can generate addresses and control signals for the WARP array. The host supplies data to and receives results from the WARP array. In addition it executes those code parts that are not mapped onto the WARP array. Each WARP cell is implemented as a programmable horizontal micro-engine, with its own micro-sequencer and program memory for 8K instructions. The cell data path consists of a 32-bit floating-point adder, a multiplier and two local memory banks for resident and temporary data, a queue for each intercell communication channel and a register file to buffer data. All of these components are connected via a crossbar.

In a typical configuration, the WARP machine consists of a linear systolic array of 10 identical cells, each of which is a 10 MFLOPS programmable processor; thus the peak performance is 100 MFLOPS. The machine can be connected to any general purpose host machine running UNIX as an attached processor. It is then accessed via procedure calls on the host or through interactive, programmable command interpreter called the WARP shell [DeCegmaA(b)89].

It is a challenge to provide a uniform address space that implements shared memory constructs with minimum access conflicts in order to maximize performance. Considerable research has been devoted to developing interconnections schemes for array processors, which allow parallel, conflict-free memory access [BalakrishnanM88]. Contemporary shared-memory multiprocessors such as the BBN Butterfly-I

(GP1000), the Butterfly-II (TC2000), and the Sequent-Balance 21000 have implemented some of the important concepts learned from this research.

The BBN systems are physically distributed-memory multiprocessors running a flavor of the MACH operating system that offers a logically shared virtual space to the programmer. This is made possible by a specially designed Multistage Interconnect Network (MIN) that allows any PE to access any memory module. The MIN, known as the butterfly switch, is the key component that provides the BBN systems with a scalable memory access bandwidth capable of supporting multiple memory accesses. A local memory access directly bypasses the interconnection network, but a remote reference is serviced via the network. Conflicts are resolved by dropping the conflicting requests and later retries. The GP1000 is built using the Motorola 68020 CISC processor, and the TC2000 is built using the Motorola 88100 RISC processor [GP88] [TC89].

The Balance is a truly shared-memory multiprocessor running the Dynix operating system. The system is built around a high bandwidth bus to which all PEs, memory modules and I/O interfaces are attached. All memory references are serviced through the bus, and multiple requests are pipelined onto the bus so that it is available for reuse while a memory access request is being serviced. The processor does not hold the bus for the entire duration of the memory access cycle. Access to hardware locks is arbitrated over a separate bus [Balance84].

2.5 MIMD Parallel Architecture Overview

The most common parallel processing approaches are: the Von Neumann-based, the dataflow-based, and the reduction-based.

The Von Neumann approach consists of interconnecting two or more Von Neumann-type processors in a variety of configurations. The main distinction is that processing takes place via a single thread of control. Having more than one processor means that there may be multiple threads of control, and the overall system control is a problem in this approach.

The dataflow approach executes instructions as soon as their operands are ready instead of following a thread of execution, as in the Von Neumann approach.

The reduction approach consists of performing instructions when results are needed for other calculations. Programs are viewed as nested applications and execution precedes by reducing the innermost applications, according to some semantics, until there are no further applications to reduce.

There is yet another approach called hybrid which combines the dataflow and reduction functionalities. In this approach processors will work first on the instructions that have been demanded of them if the operands for such instructions are ready. If operands are not ready then the processor will demand them from other processors and work on lower priority items. This results in a more balanced use of the resources because instructions are performed in order of importance.

One of the key elements in designing a general purpose parallel architecture is that users want to use the extra compute power but they are not interested in how such performance improvement is delivered. As a consequence, the new parallel systems must be able to appear completely transparent to the user.

MIMD architectures use a variety of interconnection mechanisms. Some use a bus structure and handle all processors and memory modules from the bus. [6] Data transfer operations are controlled by an I/O controller module that acts as a bus arbiter to resolve access contention problems. Due to bus conflicts, memory access cycles will be increased, which results in lower performance. A way to aid in bus contention problems is to provide the processors on the bus with some amount of cache so as to reduce their dependence on the bus for data. The only problem here is that the distributed cache subsystems have to be kept synchronized. Cache synchronization consists of the correct read/write sharing of replicated data. This implies that processors must have exclusive rights to writing shared data elements. Processors must also be able to get the latest version of a shared data item and they must be blocked from getting transitionary data elements. If the required data is available, multiple read accesses should also be allowed. Most of these implications can be implemented in software; however, there is a need for hardware support by means of flags and counters for the memory objects that are shared [BitarP86].

An alternative to the bus MIMD is the crossbar switch system interconnection, which provides the best performance of any interconnection system at the expense of complexity, size, and cost. The crosspoint switch must have major hardware capabilities such as including switching parallel transmissions and resolving multiple requests with different priorities for access to the same memory module. A representative system is the CMU C.mmp system, which has 16 processors and 16 memory modules [FullerS78].

The interconnection network of an MIMD machine can be either single-stage or multistage. The single-stage network can be viewed as an interconnected set of N input units and N output units. The allowable interconnections are determined by the way the input units are connected to the output units. There are 4 different ways to connect single-stage units: mesh, cube, shuffle-exchange, plus-minus 2^i. The

---

6 At the present time there are commercially available systems that offer this type of configuration. Vendors include Encore Computer's Multimax, Sequent Computer's Balance, Alliant Computer's FX/8, etc.

plus-minus 2^i network is a superset of the mesh network because it connects any given address X to address $X + 2^i$ and $X - 2^i$. A popular basic MIMD structure is the multistage interconnection network. It consists of multiple stages of intelligent switches capable of providing at least one path between any two system elements (processor, memory modules) with minimum complexity and cost. Multistage networks are built from stages of the basic single-stage networks [MapplesC85].

Processors in loosely coupled systems (LCS) communicate by exchanging messages while tightly coupled systems (TCS) have their processors communicate via shared memory variables. LCSs use local memory, and if a processor requires access to data in another processor's memory, it must request the data via a message. It is up to the message receiver to send the requested data, thus there is a large message-passing overhead involved in this data access mechanism, because data cannot be accessed deterministically due to the message passing delay. The process of getting the required data item may take any number of tries before the data is actually made available to the requesting PE. Every message generated has to travel through the communication subsystem; if such a communication scheme is not well-structured and controlled, this medium may become the system bottleneck.

TCS provide a global shared memory system which is accessible to any processor. Such a system may be organized as a single memory module that is connected to a bus-like medium with local cache modules to improve performance. Another organization may put local memory modules with its own secondary interconnection mechanism to make the memory space look like it is consolidated when in reality it is totally distributed. Each PE and its memory are located at a node. Collectively this configuration may actually form a shared memory system; thus any processor can access any other processor's memory by using the secondary communication mechanism. [7] From the programmer's point of view the only real difference in memory access to another processor's memory is the response time. The major advantage of communicating via shared memory is an almost unlimited buffer for asynchronous IPC. At the same time, the programming model for the machine becomes much easier and this allows higher hardware utilization. However, the overhead incurred by the secondary interconnect network can be sometimes out of reach, except for very specialized applications. Using just one interconnection network for both memory and processor communication leads to the usual access conflicts and thus performance degradation.

The basic dataflow concepts were developed in the 60s mainly by compiler experts who used dataflow graphs as tools to optimize sequential programs. A dataflow graph is a directed graph in which the nodes represent primitive functions, such as add and multiply, and the arcs represent data dependencies between

---

7 BBN Advanced Computers has already announced a commercial TCS computer with a butterfly interconnected memory module which makes the memory appear as a monolitic shared memory module. This new system runs MACH 2.5, a message passing OS, and no longer requires a front end to communicate with users

functions. In the 70s it was realized that if the dataflows were executed directly in hardware the architecture would be massively parallel. These machines are thus language-based; the architecture implements the formal behaviors of the program graph and the compiler translates the source code into the architecture's equivalent program graph. When the machine is in an execution stage, it is activated by the presence of an operand value in each operand field. The content of the execution template (instruction) defines the operation packet (result) <opcode, operand, destination>. Such an operation packet specifies one result packet with the information <value, destination> for each destination field of the template. When a result packet is delivered, the result value is placed in the operand field defined by its destination field. To this date, no commercial dataflow machine has ever been constructed. There have been several research projects, but critics argue that they will never really make it out of the laboratories because the designs have trouble dealing with large arrays and incur an excessive computational overhead. Another problem of practical importance is that debugging both the hardware and software on such machines is still not well understood [SrinivV86].

2.6 Systolic and Wavefront Arrays

A synchronous array of parallel processing elements under the supervision of one or more control units is called an array processor. In most known designs array processors are SIMD. In some cases all PEs will get the instruction and execute it; in other cases PEs will be selectively activated to execute the current instruction. All PEs not selected to execute the instruction will then be inactive for this execution cycle. The array processor is usually interfaced to a host computer through the CU. In most cases the host computer is a general-purpose machine whose function is to manage the array processor resources. The array CU directly controls the execution of programs on the array and the host computer takes care of the I/O functions required to complete the task.

It is of paramount importance to be able to transfer data among memory modules after computational steps. The number of data transfer steps must be minimized by lowering the possibility of memory access conflicts. In message-passing array processors, the interconnection network is made up of interchange boxes that are capable of setting themselves up dynamically to establish a desired connection. This is done by examining message header packet fields. These fields must be setup by the compiler; however, this type of machine level control may require the compiler to know detailed information about the computational and data transfers of the particular application. The application domain of array processors covers image processing computer vision, nuclear physics, structure analysis, speech, sonar, radar, seismic, weather, astronomical, medical signal processing applications, etc. [SchendelU84].

An interesting array processor is the systolic array, which consists of a set of simple processors interconnected in a regular manner to perform a simple instruction. The interconnection is usually static and the PEs are not general purpose; instead they perform hard-wired instructions geared towards a particular application. The PEs in a systolic array are typically connected in the form of a pipeline, array mesh, triangular, etc., and communication with the outside world occurs only at the array periphery. The basic principal in systolic arrays is that the standard CPU can be replaced by a systolic array. Therefore, this can increase the processing power without increasing the memory bandwidth.

A problem with systolic arrays is that cell synchronization in very large arrays requires long delays between clock signals due to clock skew problems. At the same time, the synchronization of large data transfers leads to large current surges as the array elements are simultaneously energized or change state. A simple solution to the synchronization problem is to make the arrays data-driven or asynchronous; these arrays are called wavefront arrays [KungH82].

The main difference between systolic and wavefront arrays is how their particular elements are synchronized. In a wavefront array, the information between PEs follows a simple protocol. Whenever data is available, the transmitting PE informs the receiver PE, which accepts it when it is ready. Once accepted, the receiver communicates with the sender to acknowledge that the data has been used. The scheme can be implemented by a simple handshaking circuit and ensures that computational wavefronts propagate in an orderly manner without crashing into one another. The biggest advantage over systolic arrays is that since there are no clock delays, a wavefront array is scalable, exhibiting a linear increase in performance as the array size increases.

Programming wavefront arrays involves the definition and assignment of computation to each PE. Systolic arrays require in addition scheduling computation. Fault tolerance is implied in wavefront arrays because if one PE fails, all subsequent computation that depended on the faulty PE will stop automatically. In order to perform the same type of fault tolerance, systolic arrays must make use of global error-halt signals, which is not desirable because of the added wire area. This particular feature makes wavefront arrays good candidates for Wafer Scale Integration (WSI), where one may be able to reroute around the faulty PEs.

The massive concurrency in systolic/wavefront arrays is derived from pipeline processing, parallel processing, or both. Although most of the current array processors stress only word-level pipelining, the new trend is to exploit the potential of multiple-level pipelining (i.e. at the bit-level, word-level, and array-level granularity) [KungH78].

Systolic arrays and wavefront arrays are characterized by inflexible and highly dedicated structures. Hard-wired dedicated processors offer high processing speed but suffer from long redesign time as one attempts to accommodate for new algorithmic requirements. With the advent of modern algorithm/ architecture analysis, the programmable array processors will become not only more economical but also more appealing in coping with constant changes of system specifications [KungS85].

A very desirable quality of array processors is reconfigurability. Here the term is used to mean the ability to alter the interconnection patterns between the PEs for certain intended applications such as multifunction or fault tolerance. Two types of reconfiguration strategies are applicable: Static reconfiguration is used to establish a preprocessing step wherein the network is configured prior to the initiation of the tasks, and dynamic reconfiguration is used to reconfigure the execution paths during run time. This latter capability is particularly desirable in applications where the communication patterns are non-deterministic. The choice of either static or dynamic reconfiguration is constrained by application-specific goals, such as real-time response, reliability, or both [YungH88].

[KorenI88] proposes a data-driven programmable array processor, which was designed and fabricated with good results. DFG algorithm mapping techniques and performance results are also presented. Much of the other research on wavefront arrays has concentrated on the scalability and synchronization, but neglected the general purpose PE and reconfigurable connection concepts. [KungS88] presents work that suggests programmable systolic and wavefront arrays are being researched, but the inter-PE connections are still static.

2.7 Asynchronous Design

Asynchronous design techniques have the advantage of being free from the lockstep constraints. At the same time they map very well the distributed computation system requirements and even provide embedded self diagnostic capabilities. Synchronous systems operate on a central clocking scheme where the clock period must be greater than the slowest combinational path possible during a clock cycle. This is in contrast to asynchronous systems, which operate at varied computational rates. Asynchronous systems synchronize their PEs by a special synchronization circuit that is local to each PE. Delay-insensitive circuits are defined as circuits that implement the synchronization function by a logical behavior which is independent of the driving module and wire delays.

The protocol definitions presented in [BrozozowskiJ89] describes a four-phase or two-phase signaling protocol used to synchronize two or more systems. In order to use such type of protocol two or more systems are connected via a synchronization channel that consists of two signals request (req) and acknowledge (ack). The end of the channel that initiates the synchronization is called active and the other

end is called passive. The four-phase protocol at the active end is as shown in eq 1 and the one at the passive end is shown in eq 2:

- req^+; $\underline{ack}^+$; req^-; $\underline{ack}^-$ (eq 1) [8]
- $\underline{req}^+$; ack^+; $\underline{req}^-$; ack^- (eq 2) [9]

These two equations describe the complete four-phase signal protocol, which is sometimes called the return-to-zero protocol. Note that the last two signal representations in each signal sequence are bringing the req and ack signals back to zero after the computation has taken place. In the two-phase protocol, the interface circuits skip the return-to-zero phase (the underlined signal identifies a wait)

Asynchronous circuit design has a number of problems which must be properly addressed. One condition that may arise in delay insensitive circuits is transmission interference. This occurs when the active end of the circuit places two conflicting signal levels on its channel. For example a req^+ followed shortly by a req^-, the passive end of the circuit would get confused and malfunction because of unstable synchronization signal values.

Another problem is signal ordering. Modules must be capable of receiving their synchronizing signals in any particular order, proper computation must not be tied to signals arriving only in a specific order. This is due to possible transitional delays induced on signal lines. For example, the passive end expects its synchronization to be done by signal a^+ followed by signal b^+ and both signals are properly generated by the active end. Yet, due to extra delays on the wire the signal order gets reversed. The result would be that the passive end would not work properly. If signal sequencing must be used, then all possible signal order combinations must be allowed.

A synchronization problem may also arise if the active end sends inputs to a passive end that is not ready to utilize them. If the previous inputs were not properly latched, then one computation step will interfere with the next computation step by contaminating the next step's input data. This problem is called computational interference because there is a mixing of current and previous inputs. The way to overcome this problem is to have the inputs latched by a signal sequence that implies that the current computation is finished and that there are more inputs to be latched, much like the protocol shown in equation 2 [GopalakrishnanG90].

---

8. The active side sets the req signal high, then it waits for the passive side to make the ack signal have a low to high transition Once the ack signal has been received by the active side it sets the req signal low and waits until the passive side makes the ack signal have a high to low transition

9 The passive side waits for the active side to make the req signal have a low to high transition Once the req signal has been received by the passive side it sets the ack signal high, then it waits for the active side to make the req signal have a high to low transition Once the req signal has been reset by the active side the passive side sets the ack low

2.8 The CHiP Architecture

The CHiP architecture is of particular importance to this thesis due to its relevance. We refer back to this architecture in later chapters as we describe our work. Some of the PAAP architectural features are based on concepts introduced by the CHiP architecture. Only the most important architectural highlights are included here, for a more complete description refer to [SnyderL82].

The CHiP is a loosely coupled MIMD architecture composed of a collection of homogeneous PEs placed at regular intervals in a two-dimensional lattice of programmable switches. Each PE is capable of performing floating point operations, and has its own local memory bank for programs and data. There is no concept of a global memory, and access to secondary storage is accomplished via the perimeter of the lattice. Switches are used to connect PEs in different ways, each switch has enough memory to store several different pre-loaded configurations such as binary-tree, hypercube, mesh, etc.

Large computational problems are implemented on the CHiP by decomposition into a sequence of parallel algorithms called phases. Each phase is described by a graph, which is directly implemented on the computer by programming the switches; the processes are implemented by writing sequential programs for each PE. The ability to configure the parallel machine dynamically to match the algorithm topology enables this type of computer to be a universal parallel computer [SnyderL81] [KungH82].

The CHiP approach to general purpose MIMD computation was shown to be feasible. The main idea behind it was that at different times during the execution of a parallel algorithm, processors may need to communicate in a variety of ways. For example, during one step a tree configuration may be needed to sum a set of numbers, in another step a rectangular mesh may be required to execute a systolic-like operation. [GannonD86] shows how the SIMPLE computation algorithm, which can greatly profit from the flexible routing mechanism provided, can be mapped onto the CHiP architecture. [10]

The CHiP's switch provides a set of predefined routing configurations. This feature turns into a restriction when a new or partial switch configuration that is not in the predefined routing set is desired.

Each CHiP PE implements a loosely coupled memory model because it has access to its own local memory. Data and programs are first loaded into the local PE memory and fetched as they are needed. This implies that the PEs may work independent of one another, but memory access cycles have to be synchronized, which limits the design's scalability. At the same time, no provisions are made for fault tolerance, and its implementation would require extra circuitry.

---

10 The SIMPLE algorithm simulates the flow of a pressurized liquid as it moves inside a spherical shell.

Chapter 3 PAAP Architecture Overview

The PAAP array proposes a methodology for mapping high-level computation into hardware structures. This feature is shared with systolic and wavefront arrays; however, the PAAP architecture differs from standard systolic or wavefront arrays in that its PEs can be dynamically programmed and interconnected via programmable REs. In contrast, systolic and wavefront arrays have static interconnections and implement only a single special purpose hard-wired instruction.

In Flynn's taxonomy, the PAAP architecture would be classified as an MIMD because each PE can execute its own instruction in an asynchronous autonomous manner, and data can flow in parallel to the PEs; data may flow as initial data and partial or fully computed results. A further MIMD classification places the PAAP array as a tightly coupled MIMD architecture because each PE does not have its own local memory, and relies on a host processor for both its instructions and data. The only problem with this classification is that Flynn's MIMD taxonomy does not fit well on array processor architectures, and the PAAP array is a wavefront-type architecture. [KungS85] discusses the different features that qualify wavefront arrays; however, there is no real definition for arrays that can be built using the PAAP architecture as a building block. Based on Duncan's taxonomy, the PAAP architecture classifies as a new type of MIMD Paradigms, the Configurable-Routing-Programmable-Element Wavefront Array (CRPE Wavefront Array).

The PAAP array addresses the 3 main asynchronous circuit problems by design. Our asynchronous approach uses a modified two-phase signal protocol which allows the active and passive end of the circuit to implement their own return-to-zero synchronization upon the receipt of the proper protocol sequence.

Transmission interference is not a problem because both the active and passive sides of the circuit depend on each other for resetting their synchronization signals. Once a signal has been set by the active side, the active side will wait until it receives its acknowledge before it attempts to modify the value of the protocol signal. Signal ordering is not a problem because all required synchronization signals are buffered upon arrival, thus computation only takes place when all required signals are present. Computation interference is addressed by buffering all of PE. The output synchronization mechanism makes sure that outputs are only latched in at the appropriate times in order to guarantee data consistency.

3.1 Introduction to the PAAP Array

A PAAP array is built by arranging 8-bit PEs in a regular lattice interconnected via 8-bit polymorphic REs as shown in figure 3. Each PE and Routing Element (RE) can be programmed to execute its own instruction, which makes the PAAP design an MIMD-type array processor. The PE implements an integer RISC-like instruction set. Programming examples in a later chapter, will show how the PE almost supports

high-level constructs without an assembler or a microcode decode stage. The RE implements a bidirectional-2-in-to-2-out instruction set. [11]

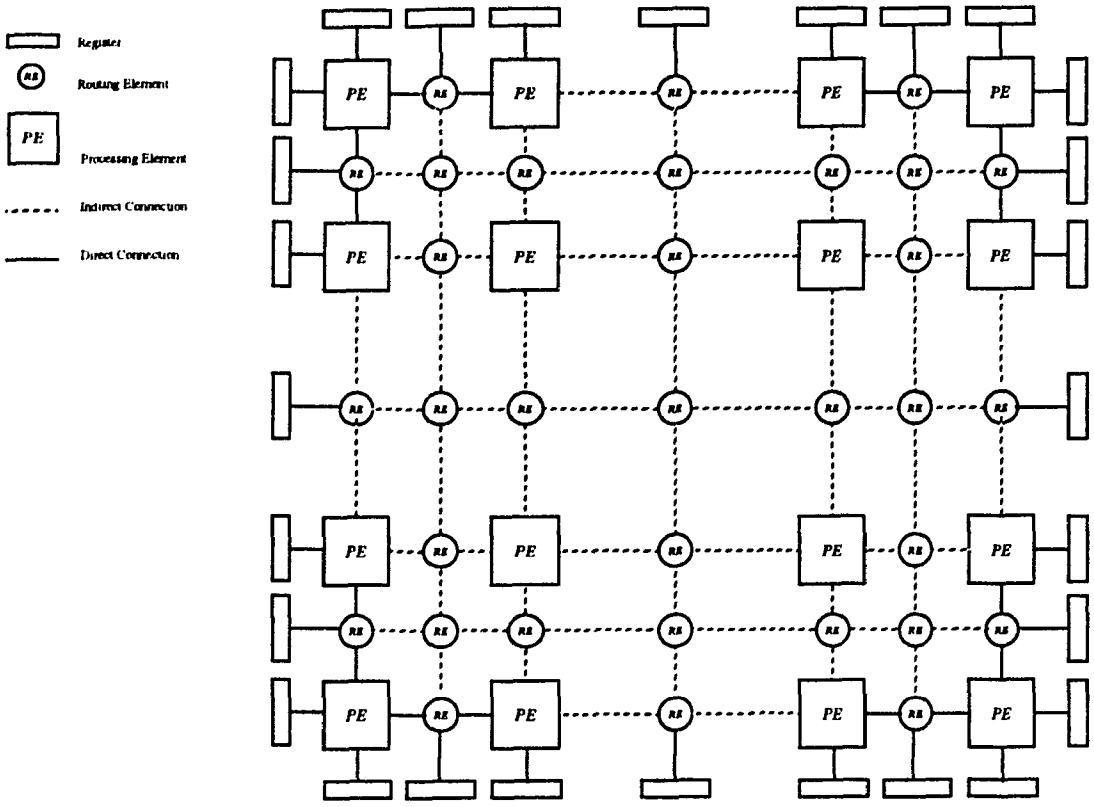


FIGURE 3. A PAAP Array [12]

The RE's functionality is based on Snyder's switch lattice concept, but its implementation is original. This PAAP RE improves on the CHiP's switch predefined routing configurations restriction by allowing individual port direction and source to target controllability. This means that the compiler can build any switching network it requires and is not restricted to H-tree, Binary-tree, Mesh, etc. configurations. Any RE port can be programmed to be an input or an output, and can flow to any one of the remaining ports. The switch implementation is novel and unique to this work.

The PE implements an original programmable wavefront array element, as such, instructions and data are not fetched from memory as in the CHiP design. Instead, each PE and RE is programmed to execute the desired instruction and routing scheme respectively during the program load phase and data flows through the complete array in wavefronts during the program execution phase. The PE provides built-in

---

11 This strategy allows any two buses to become either an input or an output. Once their direction has been chosen, then the buses can be connected to any other uncommitted input or output buses.

12 The indirect connection signifies the existence of many more PEs and REs that are not shown.

fault tolerance without incurring extra hardware overhead. The host should be able to dynamically map-out groups of PEs that do not reply to request signal chains and reroute around them. However no real hardware support for this feature is provided for it is an inherent capability in the reconfigurability of the REs and the asynchronous timing of the PEs.

The PAAP array implements a tightly coupled memory model and the shown periphery registers serve as buffers for the host to provide input data to the array and to receive calculated results from the array. The PAAP array is truly scalable because it does not suffer from clock skew synchronization problems, which may limit the array size.

3.2 PAAP Array Building Blocks

We envision a PAAP array being built by using a combination of the PAAP building blocks as shown in figure 4. These building blocks should be fabricated as separate ICs as follows:

- PAAP1 targeted to be a member of the array which provides RE capabilities on all of its output busses. This would allow us to tile the PAAP1 and create an array of any size with ready to use inter-PE connections (see figure 5). However, if this were the only IC we could use, there would be extra REs on all of the periphery output busses, which may or may not be desirable.
- PAAP2 targeted to be a periphery array element. It would be used to form the left-most column or bottommost row of the array.
- PAAP3 targeted to be used in conjunction with the PAAP2. This IC would be used to add routing elements in between the bottommost row elements. As a result the bottommost row elements will have inter-IC routing to the left. The IC would also be used to add routing elements in between the left-most column elements. As a result the left-most column elements will have inter-IC routing to the bottom.

The PAAP1 IC would use 24, 10-bit buses, therefore it would require 240 pins, plus 2 pins for gnd and 2 pins for vdd. This totals 244 pins, which can now be achieved in some advanced PGA packages to provide us with full parallel I/O to/from all busses. The PAAP2 IC would use 204 pins and the PAAP3 IC would use 124 pins. [13]

A PAAP array that contains 1089 PEs would be build as shown in figure 5. Notice that the first 10 rows from top to bottom are composed of 10 PAAP1 ICs arranged from left to right. The right most column is composed of a PAAP2, which provides the right side of the array with connections to periphery

---

13 We need two separate vdd and gnd pins, one of each for the logic and one of each for the pad rings. This helps reduce the signal noise level

PEs. In order to provide inter-row routing for all right-most column members, we use the PAAP3 element. The bottommost row is composed of a PAAP2, which provides the bottom side of the array with connections to periphery PEs. In order to provide inter-column routing for all bottommost row members, we use the PAAP3 element.

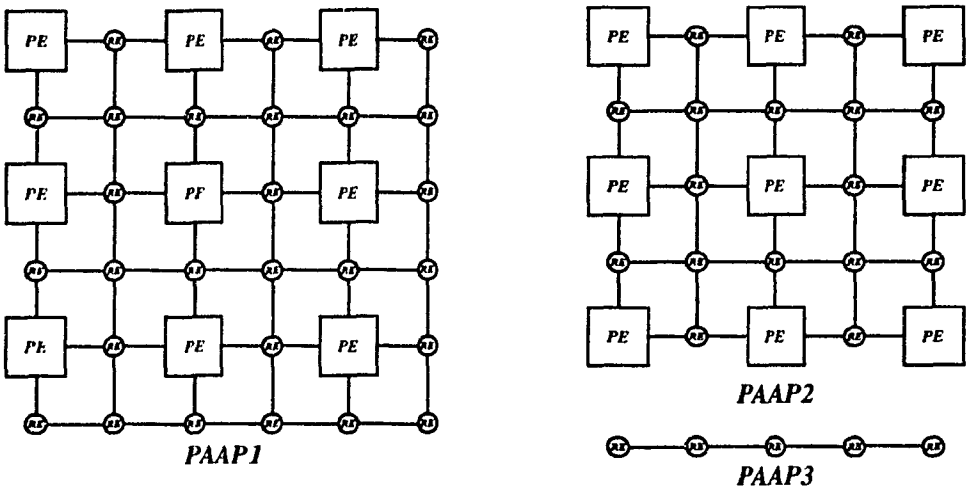


FIGURE 4. PAAP Building Block ICs [14]

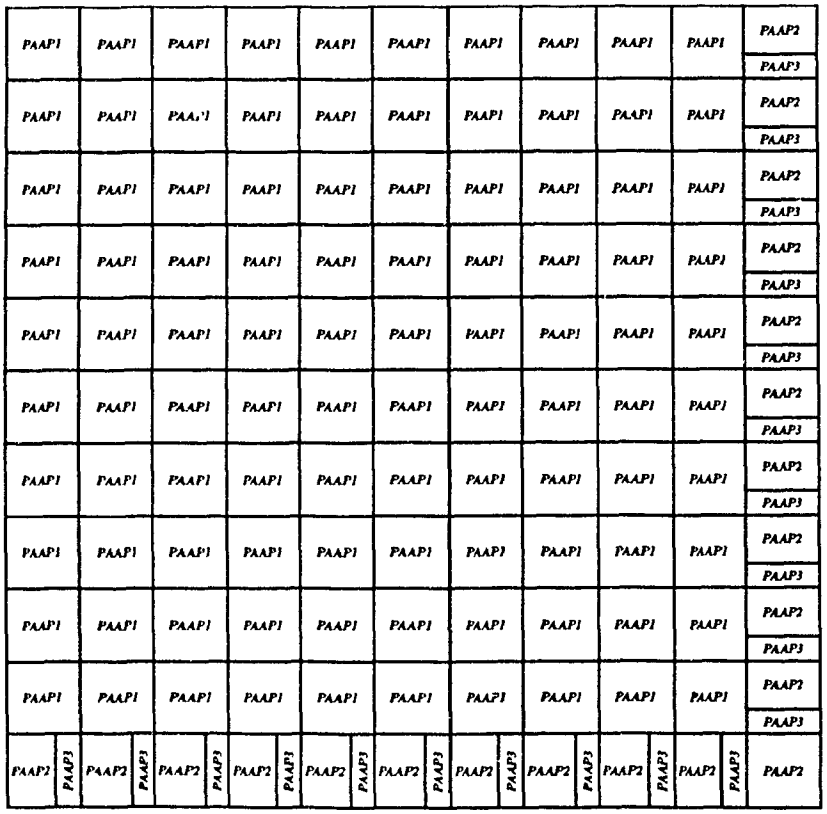


FIGURE 5. A 1089 PE PAAP Array Configuration

14 The fabrication of these building blocks is beyond the scope of this thesis

3.3 The Smallest PAAP Array Configuration

This section presents a sample array configuration as a means to show how the PE and RE can be used to form arrays. The presented ideas are possible solutions to building PAAP arrays, the decisions involved in designing an actual array are beyond the scope of this thesis.

The smallest PAAP array configuration is a single PAAP2. We will discuss how a host computer (HOST) would communicate with a single PAAP2 as a means to show the necessary interaction (see figure 6).

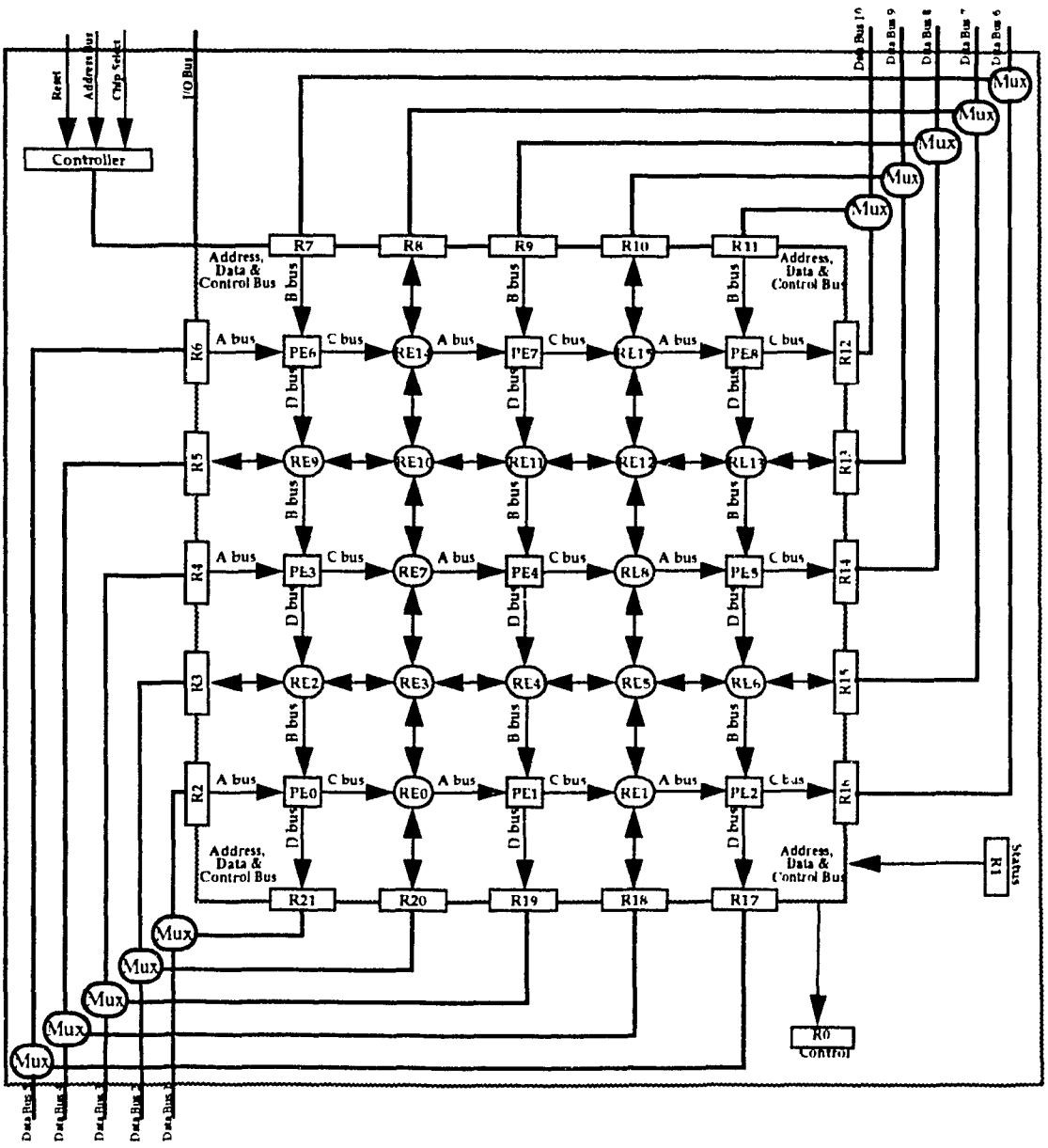


FIGURE 6. The Smallest PAAP Array Configuration [15]

A PAAP array may be build onto an extension card that can be connected to a host computer. The PAAP extension card will add the necessary logic to implement an I/O policy, as well as add the periphery registers.

A possible I/O policy could be as follows: The HOST communicates with the PAAP array by writing to its periphery registers. It sees the PAAP array periphery registers as a contiguous register file upon which it can perform I/O operations. When the HOST needs to move data into the PAAP array, the HOST provides the starting address of the memory block to be transferred and issues either a read or write signal. If the requested operation is WRITE, the PAAP array will either sample its parallel data busses or its I/O bus for valid inputs depending on the addressing style. The HOST will be acknowledged by the PAAP array once the input data has been consumed. If the requested operation is READ, the PAAP array will make sure that there are results ready for HOST access and it will provide the data along with its synchronization. The HOST will usually select a PAAP array, request the needed operation and go on to its next task. It will come back later and check the selected PAAP array's status register to determine if the requested operation has finished.

In figure 6, the processing elements are numbered from bottom-left to top-right. The bottom PE row contains PE 0, 1 and 2, the middle PE row contains PE 3, 4, 5 and the top PE row contains PE 6,7 and 8 counting from left to right. The routing elements are numbered much in the same way as the processing elements, except that there are 16 REs as opposed to just 9 PEs. RE0 is the one between PE0 and PE1, and RE15 is the one between PE7 and PE8. For programming purposes, we refer to collections of elements in terms of columns or rows. This should be intuitive since figure 6 a matrix-like structure.

Logically, the 4 I/O busses for both PEs and REs are arranged clockwise; the left side is the A bus, the top side is the B bus, the right side is the C bus and the bottom side is the D bus. For the PEs, busses A and B are strictly inputs, and busses C and D are strictly outputs. The fact that PEs have fixed locations for their I/O busses in no way limits their data flow to that of systolic or wavefront arrays. This is due to the routing element's ability to virtually redirect its busses, so the D bus output of PE4 in figure 6, could be connected to the A bus input of PE7 by properly configuring a connection path.

One may question the PE to RE ratio and the reasoning behind having almost twice as many REs as we have PEs in a single PAAP2. We could have reduced the amount of REs to 12, by eliminating RE3, RE5, RE10, and RE12. However, this would reduce the PAAP's data flow capability to that of systolic and wavefront arrays, namely, data flows only in a single direction. PEs could no longer use programmed paths

---

15 Note that the bus multiplexer, array controller, and periphery registers have been added to the original PAAP2. This allows the HOST to have full control over the array

to communicate with any other PEs in the PAAP array, they could only communicate with their immediate neighbors. We concluded that the 9 PEs to 16 REs ratio is needed in order to provide the desired system level functionality.

The PAAP array has two modes of data transfer operation:

- Address range: Where data is transferred to/from the PAAP's 10 data busses in parallel.

  This in effect allows parallel access to all periphery PEs' input or output registers. Its main purpose is to provide for maximum data transfer rates. The architectural implication is that depending on the requested I/O operation, we can read from all strictly output registers or write to all strictly input registers at once.

  A block transfer must only be done by the HOST after having checked the appropriate block-transfer-ready bit in the status register. All involved registers must be ready for the transfer. If the HOST attempts a block transfer without checking the status register, and any one of the I/O registers is not ready for the transfer, a synchronization problem may occur and the HOST will have no way of knowing about it. As a result, input data placed on the data busses could decay while all input registers are getting ready, and thus invalid data could be consumed by the array. The HOST could time-out and re-transmit, but it would be best to avoid the problem by establishing a check-first procedure.

- Single address: Data is transferred byte serially to/from the register address specified by the address contained in the address bus.

  This mode should be used when the HOST requires access to a particular register that is not accessible via a block access. The HOST would place the desired register's address on the address bus, set the chip select active high, and the read/write bit to the desired operation. The PAAP array would then decode the address and select the appropriate register for the operation. At this point, operation checks would be made to determine whether the requested operation is allowable on the currently selected register. The HOST should address the status register directly to obtain its contents, when attempting to determine if another block transfer can be performed.

The I/O registers are arranged in 3 groups (see table 3): strictly input registers, 0, 2, 4, 6, 7, 9, and 11; strictly output registers, 1, 12, 14, 16, 17, 19, and 21; and flexible input/output registers, 3, 5, 8, 10, 13, 15, 18, and 20. Each register has a tag bit that identifies what function the HOST can perform on it, read from, write to, or both. The bidirectional register's direction tag is written by controller circuit after generating

the desired register's address and checking that it is safe to overwrite its contents. The fixed-direction register's direction flag is not accessible to the controller circuit for writing. This prevents erroneous data direction settings, and helps the address decoder circuit determine when an addressing exception has occurred.

The reason for restricting the operations that can be performed on certain registers, can be better understood if we consider the periphery PE input or output synchronization. In figure 6, PE0 has its A input bus connected to R2 and its D output bus connected to R21. R2 can not reroute its output connection to any other PE's input or output bus. This by design restricts R2 to be a strictly input register. R21 can not reroute its input connection to any other PE's input or output bus. This by design restricts R21 to be a strictly output register. There is no way that the PEs or REs can modify the contents of input only registers; they are only writable by the HOST. The same applies to output only registers, we assume that they are to be written with results generated by the local PEs and REs.

TABLE 3. Addressable PAAP Registers

Address	*Register name*	*Side of the IC*	*Direction*
0	R0 Control Register	-	In
1	R1 Status register	-	Out
2	R2 is PE 0's A BUS	Left	In
3	R3 is RE 2's A BUS	Left	In/Out
4	R4 is PE 3's A BUS	Left	In
5	R5 is RE 9's A BUS	Left	In/Out
6	R6 is PE 6's A BUS	Left	In
7	R7 is PE 6's B BUS	Top	In
8	R8 is RE 14's B BUS	Top	In/Out
9	R9 is PE 7's B BUS	Top	In
10	R10 is RE 15's B BUS	Top	In/Out
11	R11 is PE 8's B BUS	Top	In
12	R12 is PE 8's C BUS	Right	Out
13	R13 is RE 13's C BUS	Right	In/Out
14	R14 is PE 5's C BUS	Right	Out
15	R15 is RE 6's C BUS	Right	In/Out
16	R16 is PE 2's C BUS	Right	Out
17	R17 is PE 2's D BUS	Bottom	Out
18	R18 is RE 1's D BUS	Bottom	In/Out
19	R19 is PE 1's D BUS	Bottom	Out
20	R20 is RE 0's D BUS	Bottom	In/Out
21	R21 is PE 0's D BUS	Bottom	Out

The strictly input register implements the proper input synchronization protocol required by the PE to which it is connected. Its implementation emulates the presence of an input partner to the connected PE.

The strictly output register implements the proper output synchronization protocol required by the PE to which it is connected. Its implementation emulates the presence of an output partner to the connected PE. The flexible input/output register has to be able to implement the required input or output synchronization protocol, based on a programmed tag setup by the HOST. This direction tag acts as a control and is assigned a value during the PAAP array program loading stage. The flexible register implementation emulates the presence of either an input or an output partner.

The only way to convert R2 and R21 into flexible input/output registers would be to connect them to an RE, located between the respective register and the existing bus connections to PE0. This design change to all restricted registers would add 12 more REs to the PAAP2, at a great cost in chip area with no real functionality gain.

An addressing exception is present in the following situations:

- If the address is invalid because it does not select a valid I/O register or block transfer.
- An address corresponding to an input register is provided and the requested operation is read.
- An address corresponding to an output register is provided and the requested operation is write.
- An address corresponding to the control register is provided and the requested operation is read.
- An address corresponding to the status register is provided and the requested operation is write.

A synchronization exception is present in the following situations:

- An address corresponding to an output register is provided and the requested operation is read, but the data contained in the addressed register is not valid yet.
- An address corresponding to an input register is provided and the requested operation is write, but the data contained in the addressed register has not been consumed.

These synchronization exceptions are considered to be temporary in nature, because they will eventually be cleared when the register becomes available. Still, they are considered exceptions in order to provide a means of communicating to the HOST that the requested register is not ready for transfer.

The PAAP array requires the following single-bit control signals to carry out all of its functions: [16]

16 Both Reset and CS signals require a falling (1 to 0) transition to be activated

- Reset: The reset signal is used primarily to reset all of the registers to a known state on power up.

  It is not advised to assert this signal while there is a program running because it will clear all PE and RE instruction registers as well as all control logic latches, along with all periphery registers' synchronization tag bit.

- CS: The chip select signal is simply used to activate all I/O operations within the PAAP array.

The PAAP array provides the user with the following 10-bit I/O busses:

- Data Bus 1 - 10: These I/O busses are meant for block data transfers to and from the PE's I/O registers.

  This allows us to load all strictly input registers in parallel or be able to retrieve data or results from all strictly output registers in parallel. The Input and Output registers are multiplexed depending on the value of the desired READ/WRITE operation bit (see table 4).

  However, this host connection is impractical for large arrays. A solution may be to arrange all periphery registers into a memory block that the host reads or writes. The memory block would have a standard memory behavior as it interacts with the host, but each memory word would either drive or be driven by the array periphery elements in a block multiplexed fashion. As a result, the array could grow without affecting the host to memory interface. Thus every time the array grows more storage locations would be added to the memory block (see figure 7).

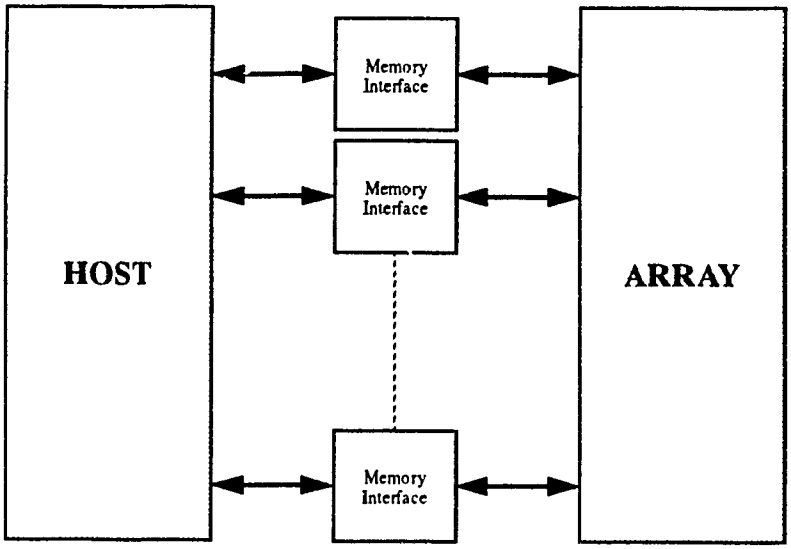


FIGURE 7. A Possible Host/Array Interface

- I/O Bus: This is a byte serial data I/O bus that can access all registers for both input or output depending on the allowable operation on the selected register.

  This bus has a special purpose when block data transfers are enabled. While doing a block read, by default it carries the contents of the PAAP array status register. This technique does not apply to the block write operation and therefore the control register does not get written to by default, because it could be destructive on already-stored data. The control register must be addressed directly.

TABLE 4. Byte Parallel Operations [17]

R/W Bit	*Data Bus*	*Register Name*
WRITE	Data Bus 1	R2
WRITE	Data Bus 2	R3
WRITE	Data Bus 3	R4
WRITE	Data Bus 4	R5
WRITE	Data Bus 5	R6
WRITE	Data Bus 6	R7
WRITE	Data Bus 7	R8
WRITE	Data Bus 8	R9
WRITE	Data Bus 9	R10
WRITE	Data Bus 10	R11
READ	I/O Bus	R1
READ	Data Bus 1	R21
READ	Data Bus 2	R20
READ	Data Bus 3	R19
READ	Data Bus 4	R18
READ	Data Bus 5	R17
READ	Data Bus 6	R16
READ	Data Bus 7	R15
READ	Data Bus 8	R14
READ	Data Bus 9	R13
READ	Data Bus 10	R12

- Address Bus: This is a n-bit address bus which provides the address of the target or source register for the requested operation. In our case n=5.

  The internal controller will make sure that the HOST is allowed to perform the requested operation on the selected register, and notify the HOST of any addressing errors via a status bit in the status register. Block I/O takes place when all bits in the address bus are set to active

17. R0 and R1 have their internal bits arranged as Least Significant Bit (LSB) on bit 0 and Most Significant Bit (MSB) on bit 7

high (11111); once this address is selected the controller will carry out a block read or block write if all synchronization checks did not encounter any problems. The two most important on-board registers are R0 and R1 because they provide the user with a simple-effective ir terface for the PAAP to receive commands for execution and for the HOST to be able to interrogate the PAAP on its current machine state.

The array implementation must include a simple way to communicate with the host. We propose a control register, which would allow the host to provide commands to the array, and status register, which would allow the array to communicate processing status information back to the host. The possible contents of the control and status registers are described in tables 5 and 6.

TABLE 5. The PAAP Array Control Register

Bit Position	*Meaning or Use*
C0	LOAD Program
C1	Re-Synchronization Request
C2	CLEAR Synchronization Error
C3	REQ Synchronization Signal
C4	Setup Pooling
C5	Read Operation
C6	Write Operation

Bit C0 is used to program the array. When this bit is active, the PE has the pass_ab instruction forced into its logic core. This overwrites the present contents of the Instruction Register (IR). The effect of running pass_ab on all PEs is that the input busses A and B are buffered and then mirrored onto the output busses C and D respectively. The RE is also forced into the straight pass switch configuration which in effect connects input bus A to output bus C and input bus B to output bus D. This bit is not destructive of the data stored in the PE or RE IRs. In order to change the value of the PE or RE IR, you must also have the appropriate bit turned active in the B bus of the column being programmed.

Bit C1 is a specific request by the HOST for a HOST-to-PAAP synchronization. It is meant to be used by the HOST to clear hanged PAAP array elements. The possibility of undetected synchronization errors arises due to our asynchronous design. Should there be a transient malfunction in any of the internal circuitry and the synchronization protocol be broken, then the PAAP array involved would be locked up. The HOST should then be able to restart the PAAP array by re-synchronizing its inputs. If the HOST makes multiple re-sink attempts and fails, it should then make a decision to signal the operating system of a faulty unit or just disable the address range in that unit. This bit is thus destructive in the sense that all registers have their synchronization-tag bit reset.

Bit C2 clears synchronization errors reported previously. However, it is specific in nature to a particular register and does not clear all synchronization as does C1. When C2 is enabled, the desired register address should also be provided on the address bus. Should both C1 and C2 be set, then C1 takes precedence over C2. If the provided register address generates an addressing exception, then this request will also generate a fatal error flag in the status register.

Bit C3 is the array global synchronization signal, it is the means by which the HOST provides global I/O synchronization to the PAAP array. It takes on a different meaning depending on which I/O operation is being requested by the HOST. If the operation is a READ, then the HOST must want to retrieve results from the output registers. This means that the PEs attached to the selected registers, block or single register address, need to obtain an ack signal. This will tell them that they are allowed to latch their results onto the output registers as soon as the results become available. Therefore, on a READ this control bit behaves as an ack signal. If the operation is a WRITE, then the HOST must be placing new data onto the input registers, therefore this signal is interpreted as a global req on a WRITE. Given that the periphery registers emulate an input, output, or input and output partner to the PE they are connected, then this req or ack signal may take on a synchronization meaning, which is particular to the I/O registers. This fact must be determined by the machine architect after the periphery registers have been designed and properly simulated.

If a byte transfer is being requested, then the selected register output is placed on the I/O bus on a READ, or the contents of the I/O bus is muxed on the selected register. If block transfer is being requested, then all input or output register req or ack signals are used to derive the global status or control synchronization bit.

Bit C4 is a request to remember the currently-provided address. Its purpose is to allow for a simple pooling mechanism. The first pooling request would then be set by properly setting the control register, register address and control signals. Later pooling requests would only require the chip select signal to be properly toggled, which would then set in motion the requested operation. Once this bit is set, the address bus can change value and its new value will be stored when the chip select signal is toggled. This mode might be used by the HOST when pooling the status register for a particular condition. The HOST sets up pooling on the status register, and then toggles the CS signal when it wants to access the status register contents via the I/O bus.

Bit C5 and C6 are the READ and WRITE operation request. The ultimate I/O operation that the PAAP array will perform is decoded by looking at both the value of the Read and Write signals and the desired address.

TABLE 6. The PAAP Array Status Register

Bit Position	*Meaning or Use*
S0	Synchronization Error
S1	Addressing Error
S2	Fatal Error
S3	ACK Synchronization Signal
S4	Processing State
S5	Block Transfer Ready
S6	Wait

Bit S0 signals that there has been an internal synchronization error. It is up to the HOST to determine how to handle the error. The notification is made and the data meant for the register in question is thrown away. There is a chance that the synchronization chain across PEs has been broken. In such case, the HOST will get a synchronization exception multiple times from the same address. The HOST should be able to determine that there is something wrong with the PAAP array and attempt to either reprogram it or re-synchronize it. This is a very crude way to implement blocking read/write operations. At the operating system level, we can only see that a read is blocking our process maybe because it is waiting for data to be available. At the micro-architecture level, the same is true, except that it is up to the HOST to keep pooling the desired address for a valid result to read. The HOST should time-out on an I/O operation after a predetermined number of tries, and thus return control to the calling process at the operating system level.

Bit S1 notifies the HOST that there has been an addressing exception. It is up to the HOST to determine how to handle the error. It could be that the HOST will choose to ignore such errors, and just keep track of how many it gets until it reaches a certain number and then raise an exception to the operating system. [18]

Bit S2 notifies the HOST that a fatal error has happened. The HOST must have been trying to read from the control register, write to the status register or an addressing exception was detected while attempting to clear a previous single register synchronization error

Bit B3 is the array global synchronization signal, it is the means by which the PAAP array provides global I/O synchronization to the PAAP array. It takes on a different meaning depending on which I/O operation is being requested by the HOST. If the operation is a READ, then the HOST must want to retrieve results from the output registers. Given that the previous results have been consumed, then this bit signals a req or valid data ready on the selected register(s). If the operation is a WRITE, then the HOST

18 The implementation of error handling policies is left up to the architecture designer

must be placing new data onto the input registers, therefore this signal is interpreted as a global ack on a WRITE.

Bit S4 signals the processing state in which the PAAP array has been set. The possible values are (active high) which means that the PAAP array is running, and (active low), meaning that the PAAP array is not running. It is a means for the host to check before attempting to restart an array element.

Bit S5 signals that the PAAP is ready for another block transfer. This is not the same as the S3 because S3 will only be generated when all output registers have valid results; however, the HOST needs to be told when it is safe to provide another block of input data so as to keep the input data flow constant. The main difference between S3 and S5 is that S3 may be signaling a condition related to a specific register, and S5 always relates to the block of registers.

Bit S6 signals that the requested operation is in progress and the HOST must wait for its completion. The HOST can use this signal for performing primitive scheduling.

Because bit S7 is not in use in this version, the status register is only 7 bits wide. This bit is always tied low for data transfer purposes.

In order to program the PAAP array, PEs and REs are configured to form two busses through the array. One of these is used to pass the instructions to the PEs and the other is used as a set of control lines to load data into the desired row of PEs or REs. This means that the instructions for an entire column of PEs or REs are loaded in parallel. Programming the PAAP array should be done as follows (see appendix A for a detailed programming algorithm):

- Set all internal registers to a known state
- Write control word to the control register
- Initialize all Input registers to 0
- For i from j down to 1
    - Set the column instructions on data buses 1..j
    - Signal column i to begin sampling its IR input bus
    - Wait for (i - 1) column propagation
    - Signal column i to finish loading the IRs

Following the completion of loading all of the registers, the load bit in the PAAP control register is returned to zero to place the array in an operational mode.

Chapter 4 The PAAP Programming Paradigm

An architecture is a hardware representation of a computing paradigm. The architecture's relevance is a reflection of this paradigm's capacity to encompass a range of computation, and the ease with which this range of computation can be mapped onto the hardware.

A Von Neumann architecture accepts the mapping of any computation, as long as it can be expressed as a sequential set of primitive operations. An array processor can accept the mapping of a set of events in parallel, but lacks the capacity to execute such operations by itself, without the help of an outside controller. Systolic and wavefront arrays are special purpose and non-programmable, and also require a host environment to provide them with data and instructions. This is contrast to the Von Neumann architecture which can fetch and execute its own instructions and data.

The computation paradigm of the PAAP architecture can be expressed as the parallel execution of a set of interacting operations. It is analogous to a set of programmable asynchronous pipelines which may interact with their neighbors. The term neighbors in this context does not necessarily imply proximity, as the programmable path mechanism provided by the RE allows any PE to connect its input or output busses to any other PE in the array. Routing is deterministic in that a connection path must be programmed along with the programming of the PEs. The PAAP is not capable of sustaining its own instruction/data stream, and hence it is not a "computer", but is a computing element which operates in conjunction with a host computer (in the same sense as an array processor or systolic/wavefront arrays).

The PAAP architecture draws its design from concepts such as programmable PEs and REs, self-timed circuits, wavefront, pipeline and array processor architecture. The novelty of this architecture is the way in which such concepts have been combined for massively parallel applications.

The PAAP exhibits an unrestricted data direction via its programmable RE. This capability is available even if the PE can only take its inputs from the right and top and place its outputs at the bottom and left directions. The unrestricted data flow direction is all implemented by the RE and not the PE.

4.1 Operational View of the Architecture

The architecture can be viewed in terms of simple and complex operations. We define simple operations of the PAAP array as the primitive operations that can be performed by its constituent elements, the PE and the RE. The PE's main computational element is a standard ALU [TTLBook85] modified to generate an asynchronous clocking signal and extended for this work to include some fundamental computing functions. The PE's instruction set provides 44 instructions, listed in table 7.

TABLE 7. PE Instruction Set

Operands	*Instruction*	*Op-Code*	*Function*
0	CLEAR	0x13	F=0
	SET	0x1c	F=FF
	NEG	0x23	F=-1 in 2's complement
1	INC_A	0x00	F=A + 1
	DEC_A	0x2f	F=A - 1
	SHIFT_A	0x2c	F=A << by 1, with 0s padding
	SHIFT_AC	0x0c	F=A << by 1, with 1s padding
	SHIFTR_A	0x3f	F=A >> by 1, with 0s padding
	NOT_A	0x10	F=~A
	NOT_B	0x15	F=~B
	PASS_A	0x20	F=A
	PASS_B	0x1a	F=B
2	OR	0x1e	F=A OR B
	ORC	0x01	F=(A OR B) + 1
	NOR	0x11	F=A NOR B
	AND	0x1b	F=A AND B
	NAND	0x14	F=A NAND B
	XOR	0x16	F=A XOR B
	XNOR	0x19	F=A XNOR B
	ADD	0x29	F=A + B
	ADDC	0x09	F=(A + B) + 1
	SUB	0x06	F=A - B
	COMP_SUBC	0x26	F=(A - B) - 1, and Status is A comp B
	PASS_AB	0x1f	F=A and status is B
	PASS_ABCOND	0x0f	F=A and status is B if cond is met, else F=0
Compound	INST1	0x02	F=A OR NOT B + 1
	INST2	0x04	F=A + (A AND NOT B)
	INST3	0x05	F=(A OR B) + (A AND NOT B)
	INST4	0x08	F=A + (A AND B)
	INST5	0x0a	F=(A OR NOT B) + (A AND B)
	INST6	0x0d	F=(A OR B) + A
	INST7	0x0e	F=(A OR NOT B) + A
	INST8	0x12	F=NOT A AND B
	INST9	0x17	F=A AND NOT B
	INST10	0x18	F=NOT A OR B
	INST11	0x1d	F=A OR NOT B
	INST12	0x24	F=A + (A AND NOT B) + 1
	INST13	0x25	F=(A OR B) + (A AND NOT B) + 1
	INST14	0x27	F=A AND NOT B - 1
	INST15	0x28	F=A + (A AND B) + 1
	INST16	0x2a	F=(A OR NOT B) + (A AND B) + 1
	INST17	0x2b	F=A AND B - 1
	INST18	0x2d	F=(A OR B) + A + 1
	INST19	0x2e	F=(A OR NOT B) + A + 1

The PE's instruction set is implemented by both the ALU and CU. The ALU implements 41 distinct instructions and the CU implements 3. The extended PE operations include the generation of a status word, conditional pass, shift to the right, and mirror inputs, along with the capability to internally route the result and status words in a switch box manner. [19]

With the extensions provided in this design, the status word from a PE may be used as input to other PEs which can control data flow based on any one of the status bits. Pipelines of PEs can implement processing segments which include conditional alternation, a fundamental property of both sequential and parallel computation. Complex computation steps can be built using a combination of shift left or right, increment, subtract, decrement, add, etc. Mirroring inputs is useful in broadcasting data and making the PE appear as a buffer. Switch box routing on the outputs allows the PE to provide an even richer instruction set for every one of the original instructions can be configured in 4 ways as shown in see table 8. [20]

TABLE 8. PE Output Routing

Control1	*Control2*	*Configuration*	*Comment*
0	0	r -> c and s -> d	Straight
0	1	s -> c and s -> d	Broadcast Status
1	0	r -> c and r -> d	Broadcast Result
1	1	s -> c and r -> d	Cross

The essence of the organization presented here is captured by the configurable routing which gives the PAAP array its flexibility and power. The RE is a four way bidirectional router which provides maximum flexibility in the definition of pipeline routes. The RE permits a pipeline to be defined in any direction across the array, downward, sideways in either direction, and upwards. In an extreme example, a pipeline can thread down and up through all the PEs on the chip. The router acts as a switch which together with other routers form I/O paths that connect PE input and output ports. The RE instruction set is dyadic in that two distinct paths are programmed onto it with each instruction. These instructions define two of the RE ports as inputs and the remaining two ports as outputs, as well as which input is connected to which output. Note that any one of the RE ports can be programmed to be either an input or output, but not an I/O port. The RE's simple instruction set is shown in table 9.

19 The added instructions pass_ab, pass_abcond, and shiftr_a allow the PE to implement data flow capabilities

20 The output routing controls are part of the PE instruction bits, control1 = IR bit 7 and control2 = IR bit 6

TABLE 9. RE Instruction Set

RE IR Bits 7 6 5 4 3 2 1 0	*I/O Bus Configuration*	*Hex Value*
1 1 1 0 1 1 1 0	a->c, b->d	0xEE
1 1 1 0 1 0 1 1	b->c, a->d	0xEB
1 1 1 1 0 1 0 1	a->b, c->d	0xF5
1 1 0 1 0 1 1 1	c->b, a->d	0xD7
1 0 1 0 0 1 0 1	a->b, d->c	0xA5
0 1 1 0 0 1 1 0	d->b, a->c	0x66
1 1 1 1 0 0 0 0	b->a, c->d	0xF0
1 1 0 0 1 1 0 0	c->a, b->d	0xCC
1 0 1 0 0 0 0 0	b->a, d->c	0xA0
0 0 1 0 1 0 0 0	d->a, b->c	0x28
0 1 0 0 0 1 0 0	c->a, d->b	0x44
0 0 0 1 0 1 0 0	d->a, c->b	0x14

A complex operation is constructed from a set of simple operations. Complex operations consist of the concatenation of PE operations in pipeline segments, and their combination through RE routing, which can join these segments in dyadic operations, broadcast outputs, or permute them through the RE/PE cross switching capabilities. The PAAP provides flow-through routing only. Iteration requires intervention by the controller of the PAAP, which may provide successive data elements through the unit, or the same data elements if the program so requires.

We define as an algorithmic unit, a step in a program which is implemented using simple operations, but can be viewed as a subprogram entity within the complex operation. It may range from the concatenation of simple PE/RE operations to any set of possible concatenations.

4.2 PAAP Programming Variants

The PAAP architecture is designed for parallel processing. This implies that data structures and computational steps be defined for parallel execution either by language constructs or by a compiler.

A program counter (PC) in a sequential architecture implies that there is a central thread of control and that the PC is keeping track of the instruction being executed. This concept is not needed in the PAAP architecture because each PE works in an MIMD-type fashion, and the host controls data flow rather than instruction flow. The computational aspect of a program is imprinted on the PAAP in the form of instructions for the PEs, and routing between the PEs programmed into the REs. The program imprint must be prepared by a compiler and loaded by a loader which runs in the host and uses the loading procedure described in Chapter 3 and appendix A.

A jump instruction implies a particular thread of control is changing from its present memory address space to a new one or just to a different location within its own address space. This notion implies a sequentiality about the algorithm because it assumes that the instruction found at the destination address will sequentially follow the instruction at the source address. Since instructions are executed as a wavefront in our architecture, there is some merit to the sequential jump instruction, but we can redefine it slightly as follows:

> In a sequential architecture a jump means that the current thread of control will be changed and that the next instruction to be executed is at the jump target address. In our case, we do not have a central control unit, and thus changing a thread control flow only involves properly programming conditional passes so that if the jump instruction is to be executed, data will flow to the new target address. Otherwise data will have its normal flow and the target address will get a zero. This means that a jump instruction in our architecture is an abstract concept that can be built out of the original instructions by properly ordering the data flow on particular address subspaces within the PAAP array.

4.3 Examples of Algorithmic Units

In any application, one must always map an algorithm onto the architecture. The design goal of the PAAP architecture is to provide a capability of having general purpose algorithms mapped onto it. This process involves the allocation of PEs and communication paths.

Even in an architecture which can perform general purpose functions, there is a certain type of algorithm that it suits best. In the case of the PAAP architecture, it is best suited for applications that require a reconfigurable PE interconnection network and an MIMD-type data flow. We may even suggest that the PAAP architecture can be programmed to implement characteristics that are attributed to the massively parallel connectionist architecture. In such a model, system knowledge is stored as a pattern of connections among the PEs so that the stored knowledge dictates how the PEs interact and thus their response to incoming data. Looking closer at how our PAAP implementation can fit into the connectionist model, we can see that programming a PAAP with all of its PE and RE instructions could be considered equivalent to the process of storing knowledge in the connectionist model. Once stored, data will be processed in a particular way as it flows through the network [FahlmanS87][AbelsonH84].

4.3.1 Pre-increment

The following high level statement can be implemented as shown in figure 8: $m = ++n$. Notice how the PE's B input is uncommitted because INCA is a 1 operand instruction. The CU adjusts the I/O

synchronization to account for the number of expected operands. The same instruction can have 4 different output configurations depending what the program requires.

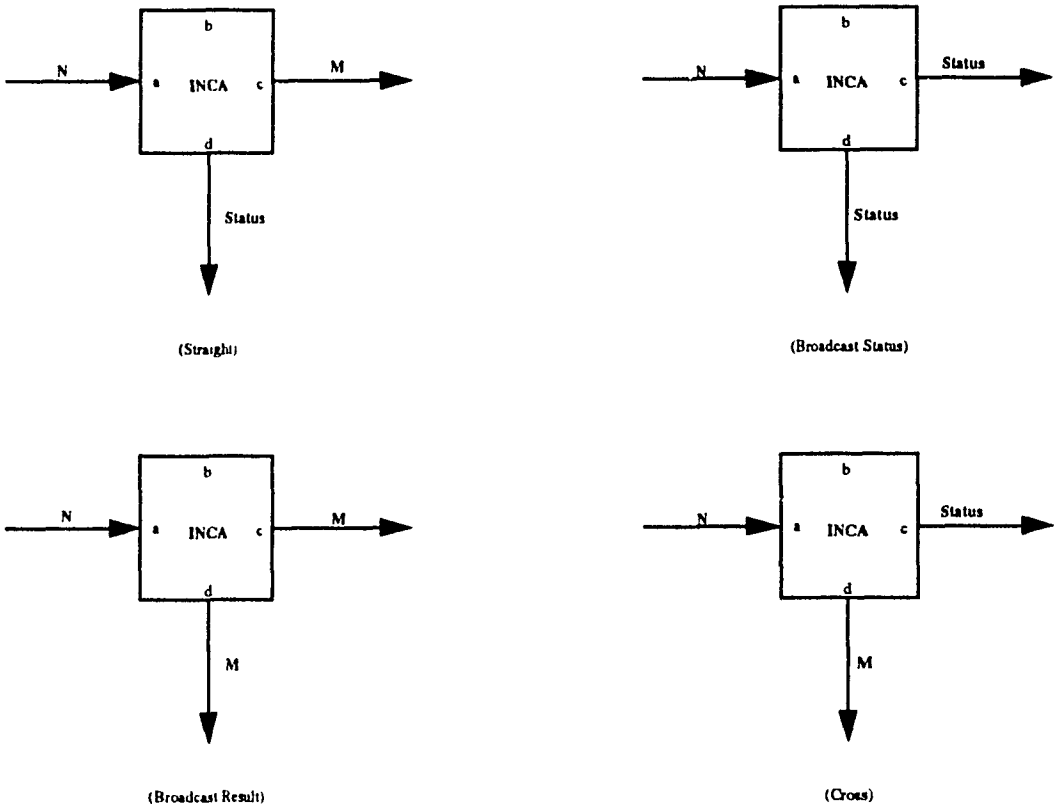


FIGURE 8. A Pre-increment Implementation

4.3.2 Swap

The following high level statement can be implemented as shown in figure 9: ***k = swap(i,j)***. In this case k is a list containing 2 elements. Using a combination of PE instruction and output routing we can accomplish a two element swap using only 1 PE. Note that the k list data structure is completely mapped onto the architecture without the use of standard storage elements.

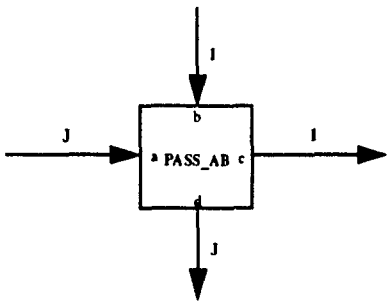


FIGURE 9. A Swap Implementation

4.3.3 Post-decrement

The following high level statement can be implemented as shown in figure 10: ***m = (~n)--***. This example shows the concatenation of three PEs into a small pipeline. PE1 negates its B input and send its output as the B input to PE2 via the shown programmed path. Notice how the ~n result travels down, to the left, up, to the right, and down until it reaches its destination. This illustrate the powerful programmability of routing paths and data flow direction control. The RE feeding PE1's B input implements the dyadic route instruction that allows n to be routed to PE1 and ~n to be routed to PE2. In a post-decrement the variable value is captured first and then the variable is decremented. PE2 mirrors its B input onto the output busses thus broadcasting it to each one of its output partners and implementing the variable capture part of the post-decrement. PE3 then takes its A input and decrements it by 1. The inter-PE path needed to send n from the PE2 to PE3, shown as a thick shaded arrow, can be as simple as a single RE programmed path or as complex as necessary. The output n generated by PE3 is a different n than the input n obtained by PE1. The data storage statement implied by the "=" has in a sense been mapped onto the architecture, for every time the input n changes, the output n will change with it. If in a sequential algorithm, a step requires the value of n before the post-decrement statement, then the input n would be provided, should the value after the post-decrement statement be needed then the output n would be provided.

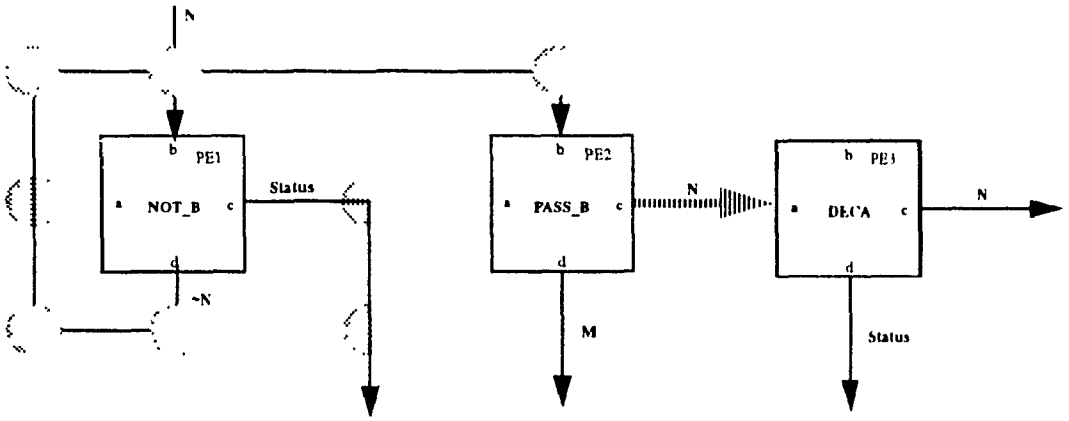


FIGURE 10. A Post-decrement Implementation

4.3.4 Arithmetic and Logic Expression Evaluation

Arithmetic and logic expressions can be evaluated by using simple or compound instructions. The following expression can be implemented as shown in figure 11: ***m = n = ((c ~| d | ~ e) - (~a | b)) & ((g | h) + (g & ~h) + 1)***. In the previous example we showed a single pipeline which exhibits a sequential behavior because all operations need to be synchronized on a single input variable, namely the variable n. The current example has 7 input variables and thus can be expressed in a more parallel fashion. Notice how data dependencies dictate that there be 3 evaluation states. In the first stage, all inputs are operated on and

partial results are generated for the 2nd stage. The second stage evaluates the first half of the complex expression using intermediate results as its inputs. The third stage combines all partial results into the final evaluation and broadcasts the result on both its outputs to map the double assignment in the expression. Compound instructions are very powerful because they allow multiple operations to be done on their inputs in once computational step. The same compound expressions can be built by using simple instructions, but require more time and processing elements.

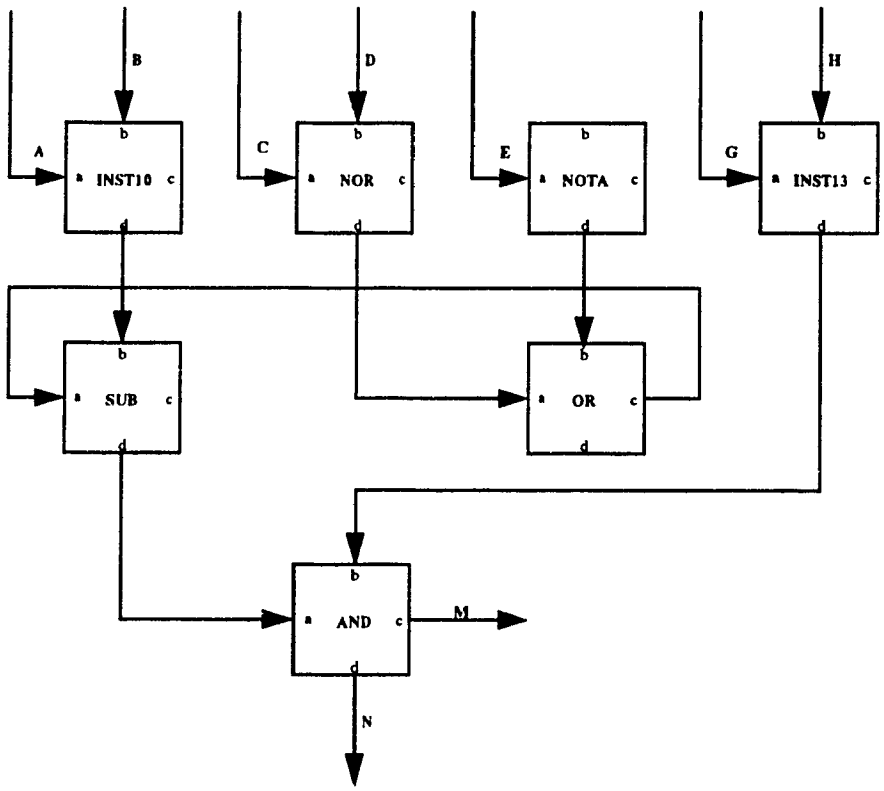


FIGURE 11. An Arithmetic and Logic Expression Evaluation Implementation

4.3.5 Inclusion of Constants in the Computation

It is of crucial importance to be able to generate arbitrary constants as part of the computation. The following expression containing constants can be implemented as shown in figure 12: i = ((j=1) + (k=4)). The set instruction can be used to initialize the constant generation at 0xff instead of 0x00. The strategy for generating constants should take into account the shortest path required to generate the desired constant values and start with a 0x00 or 0xff. Shift, dec, inc instructions can be used to create any 8-bit desired constant. The constant generation shown here is static in nature because no triggering inputs are required. In this case, the outputs i, j, k will always have the same constant value, thus the shown mapping is in fact a constant generator.

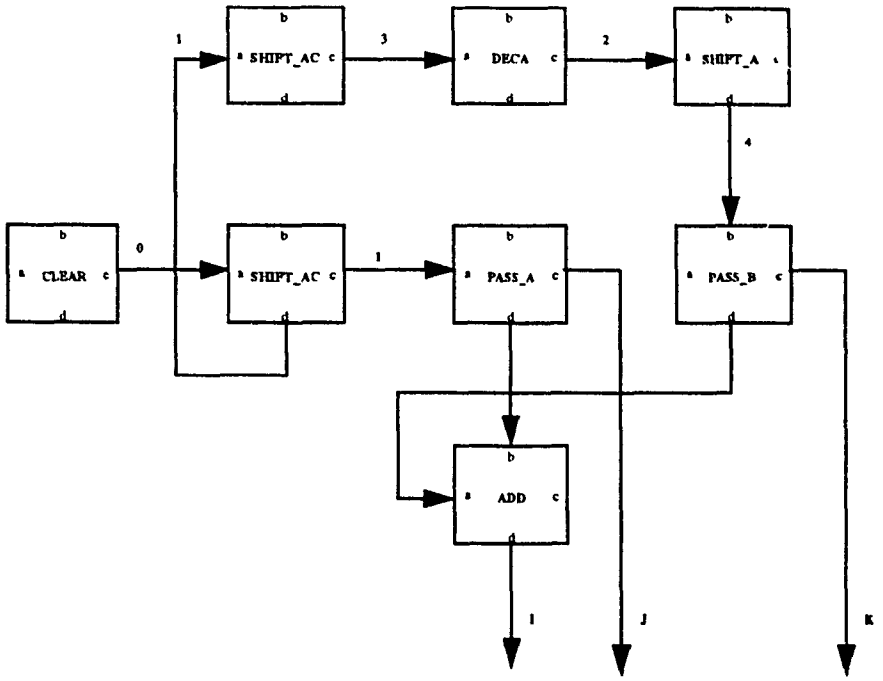


FIGURE 12. An Inclusion of Constants Implementation

4.3.6 Boolean Expression Evaluation

Boolean expressions can be evaluated as shown in figure 13: *flag = (j >= 0xfe)*. Note how the clear and set instructions are used to generate the comparison constant and the boolean operand.

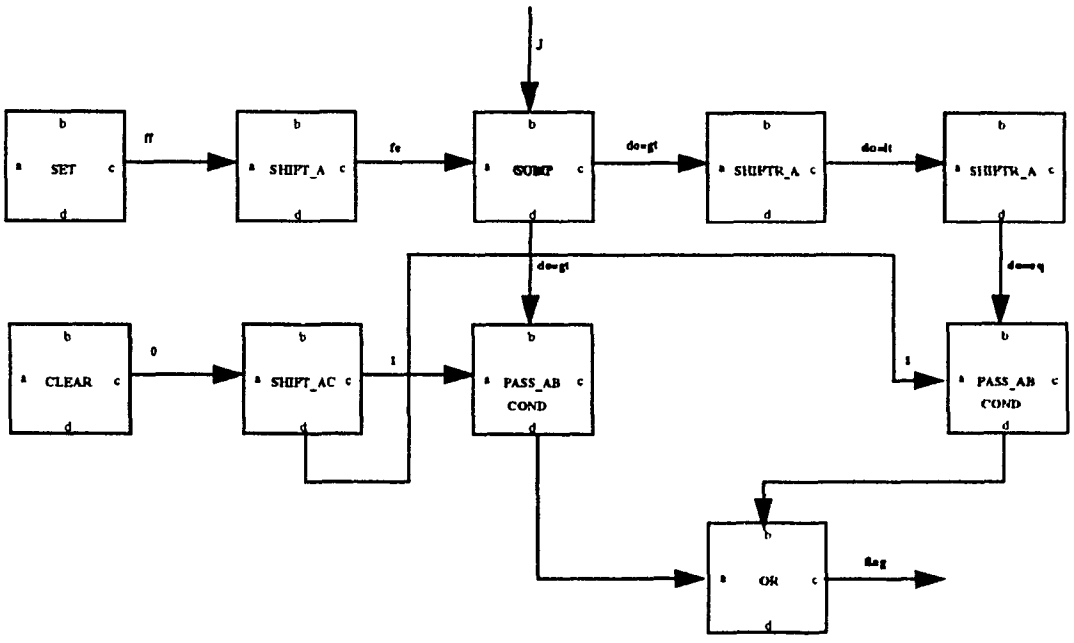


FIGURE 13. A Boolean Expression Evaluation Implementation

4.3.7 High Level Language IF Construct

We must remember that in this architecture, flow control refers to data flow control as opposed to instruction flow control. The following if statement can be implemented as shown in figure 14.

if (a < b) then c = a

else c = b

Note how the pass_a (broadcast A) instructions makes sure that we broadcast the input values in order to make the required data streams and thus allow for maximum parallelism. The comp_subc instruction provides us with a full comparison of its inputs, and we then shift the resulting status work to get at the desired comparison bit. Once we have the < status bit at position 0, then we pass A conditioned on the value of the < bit, else we pass a zero. This will implement the "then" branch. We then implement the "else" branch where we want the < condition negated so that we can pass B only when A is not passed. Finally we OR the results of both conditional pass instructions and provide the ored result on the output bus C.

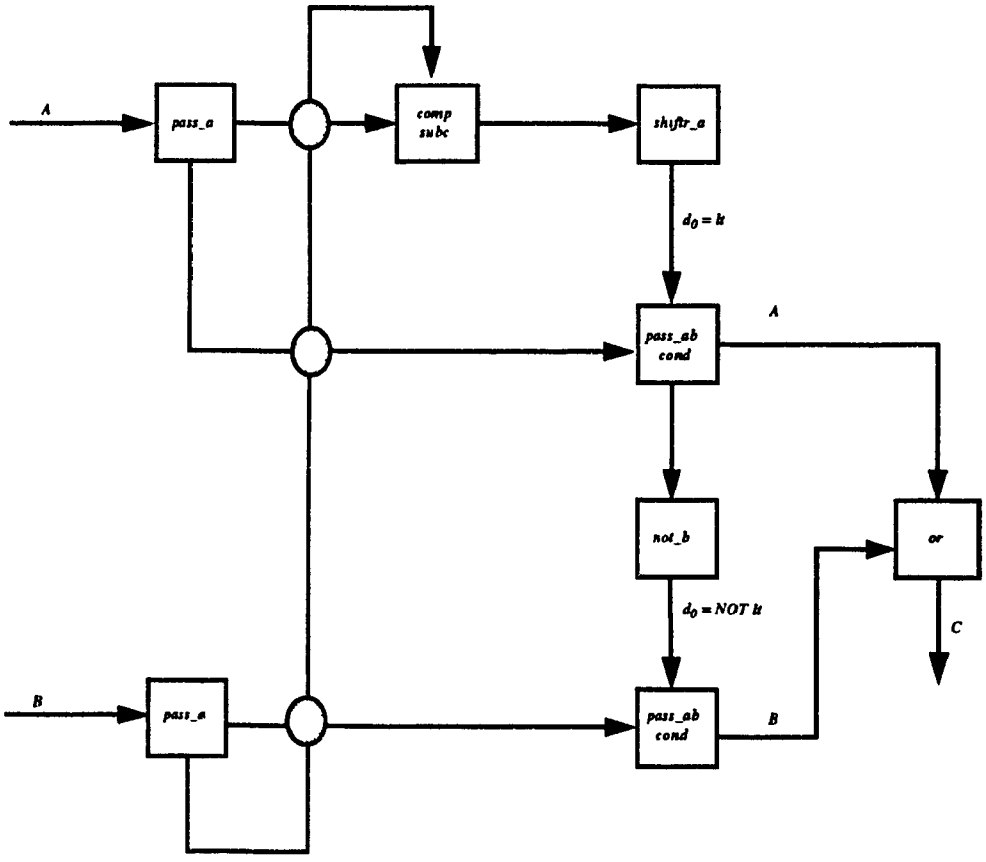


FIGURE 14. An IF Statement Implementation

4.3.8 High Level Language Looping Constructs

Looping constructs such as the "for" and "while" are not supported by the PAAP architecture due to synchronization problems in their implementation. However, a for-loop can be implemented by properly folding it, and either program a repetitive stage that generates partial results or program the complete folded loop as an algorithm. The while-loop can only be implemented via a repetitive stage that generates partial results. This is due to the fact that the occurrence of the stopping condition on a while loop can not be predicted as it can for for-loops. The following for-loop can be implemented as a folded loop, as shown in figure 15:

for(i=0; i <= 3; i++)

A = A << 1; B = B >> 1; C = A + B

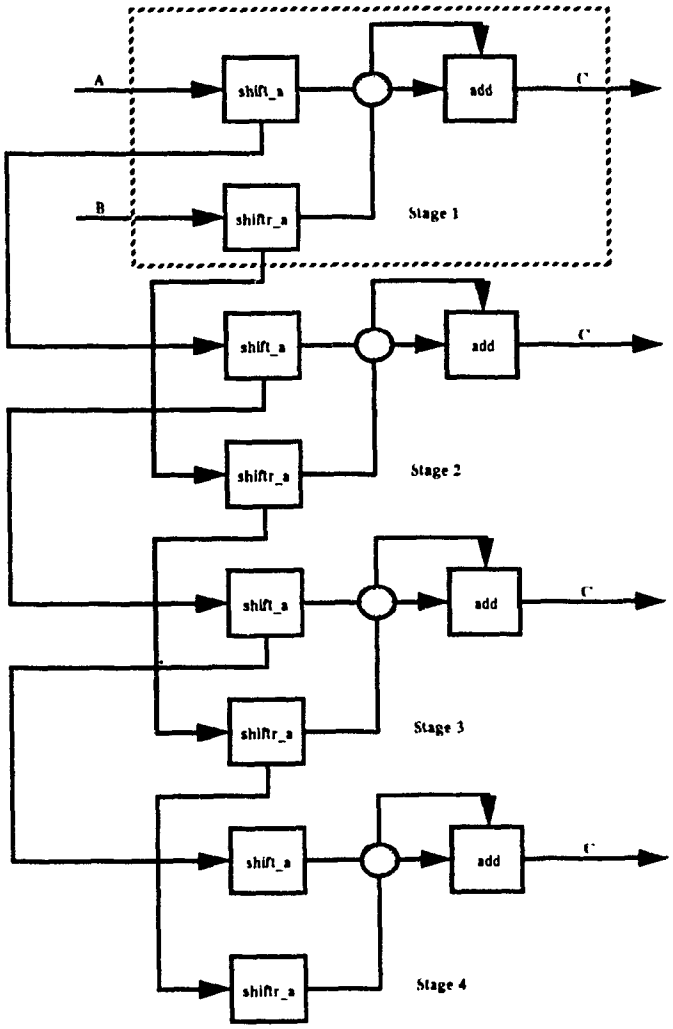


FIGURE 15. A FOR Statement Implementation

The compiler would use the starting and stopping conditions to determine how many stages to create and would synchronize them either by having them as four consecutive stages implemented in hardware so that the values of A and B that the second stage gets are the already-adjusted values from the first stage and so on. An alternative would be to build just one hardware stage, provide its inputs, collect its outputs and re-submit the previous outputs as the new inputs as many times as the loop requires. This would be a preferred implementation, but it is left up to the compiler's discretion to make that decision.

The loop synchronization problems come from the fact that our architecture is asynchronous and thus feedback loops make it almost impossible to guarantee the proper synchronization of the array. Once a seed input has been placed in a loop, it may propagate through the loop with almost no control by the host. At the same time the synchronization dependencies are recursive in nature and there is no real way to get the loop started. To illustrate the point, we examine the above mentioned folded loop as if we were to map it onto a feedback loop (see figure 16).

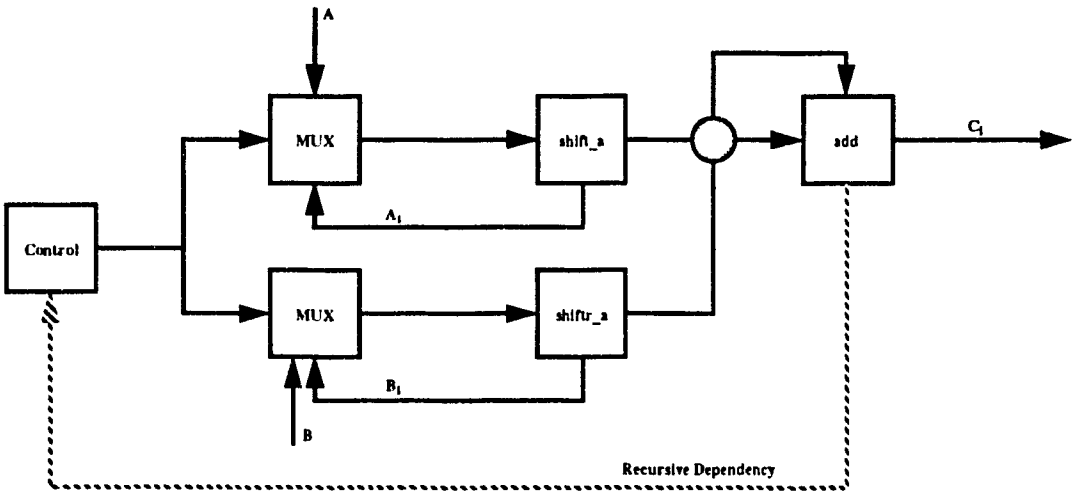


FIGURE 16. Deadlock Synchronization

We want to multiplex the input to the shift_a instructions so that they get the original value of A and B the first time and all successive times they get the shifted values of A and B. One way of doing it would be to have a Boolean flag set to false originally. This flag would control the mux to provide the original values to the shift operations; however, the flag must be reset to true so that the shifted values can be used, but the resetting of the flag depends on the a previous value already being done. However, this will not get done until it gets its inputs, and thus there is a deadlock synchronization problem.

4.4 Examples of Complex Operations

The complex operation examples will build up algorithmic units by using simple operations. These algorithmic units will then be used to map data structures and computational steps.

4.4.1 Min/Max Search Algorithmic Units

The IF statement mapping shown in section 4.3.7 is in fact implementing a Min Algorithmic Unit because the output C will always be the smallest of its inputs A, B. The Max Algorithmic Unit can be implemented by modifying the mapping shown in 4.3.7 as shown in figure 17. Note that the only change is related to which status bit we are interested.

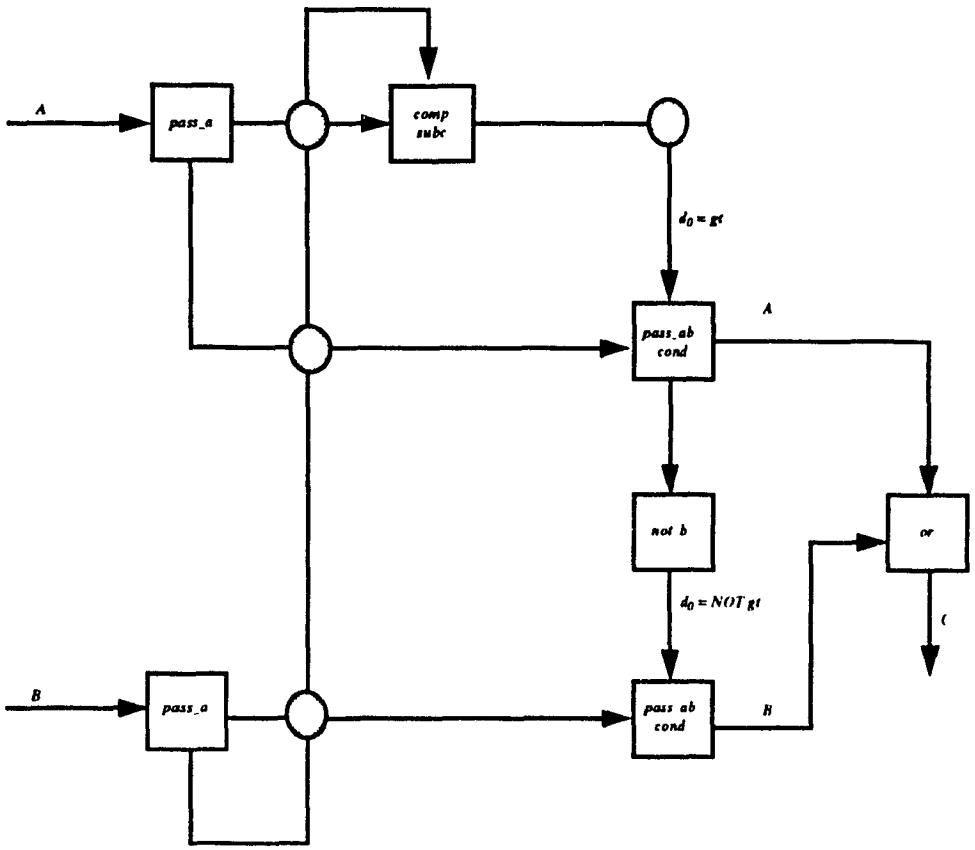


FIGURE 17. A Max Algorithmic Unit Implementation

Searching for the smallest or largest element in a list of elements is reduced to the parallel mapping shown in figure 18 and 19. The implementation shows how an inverted binary tree data structure can be mapped. This data structure allows as to implement a binary search algorithm. If we assume that each min/ max algorithmic unit takes n time units to generate its result, then the algorithm has a run time of at most depth of tree * n time units; this run time calculation assumes that all inputs are available in parallel.

Note that the mapped data structure is capable of handling a steady stream of lists composed into data wavefronts for maximum parallelism. The asynchronous nature of the PAAP guaranties that while $list_{i-1}$ is being processed by the stage 3, $list_i$ can be processed by the stage 2 and $list_{i+1}$ can be processed by stage 1.

We are thus using as many levels of parallelism as the algorithm mapping allows. In this case particular we have a 2 level pipelining implementation one at the algorithmic unit level and one at the stage level.

This example also shows how simple operations are integrated into algorithmic units, which are then integrated into algorithm implementations. The level of integration is only limited by the programmer's imagination and the availability of a large enough PE/RE pool for the needed implementation. The Min/ Max algorithms shown here can be considered as compound algorithmic units to be used as single entities in a higher level program.

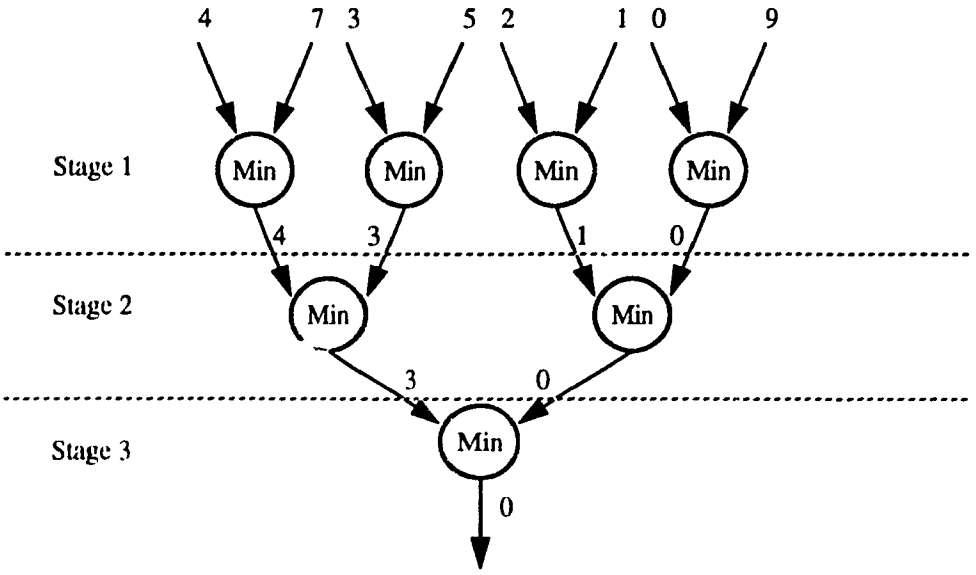


FIGURE 18. A Min Algorithm Implementation

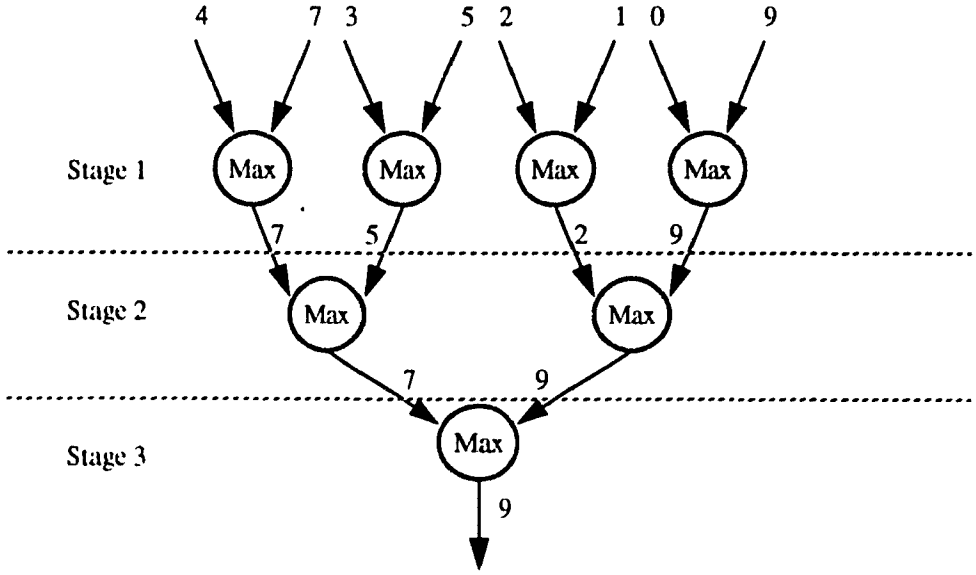


FIGURE 19. A Max Algorithm Implementation

4.4.2 A Sort Algorithmic Unit

The process of sorting involves the comparison of incoming inputs and based on their relationship (<, >, =) an action must be taken to generate the results. A descending sort step can be implemented as shown in figure 20. An ascending sort step can be implemented by simply deleting the PE that is implementing the shiftr_a instruction from the pipeline (shown shaded). Note how this algorithmic unit has two inputs and two outputs.

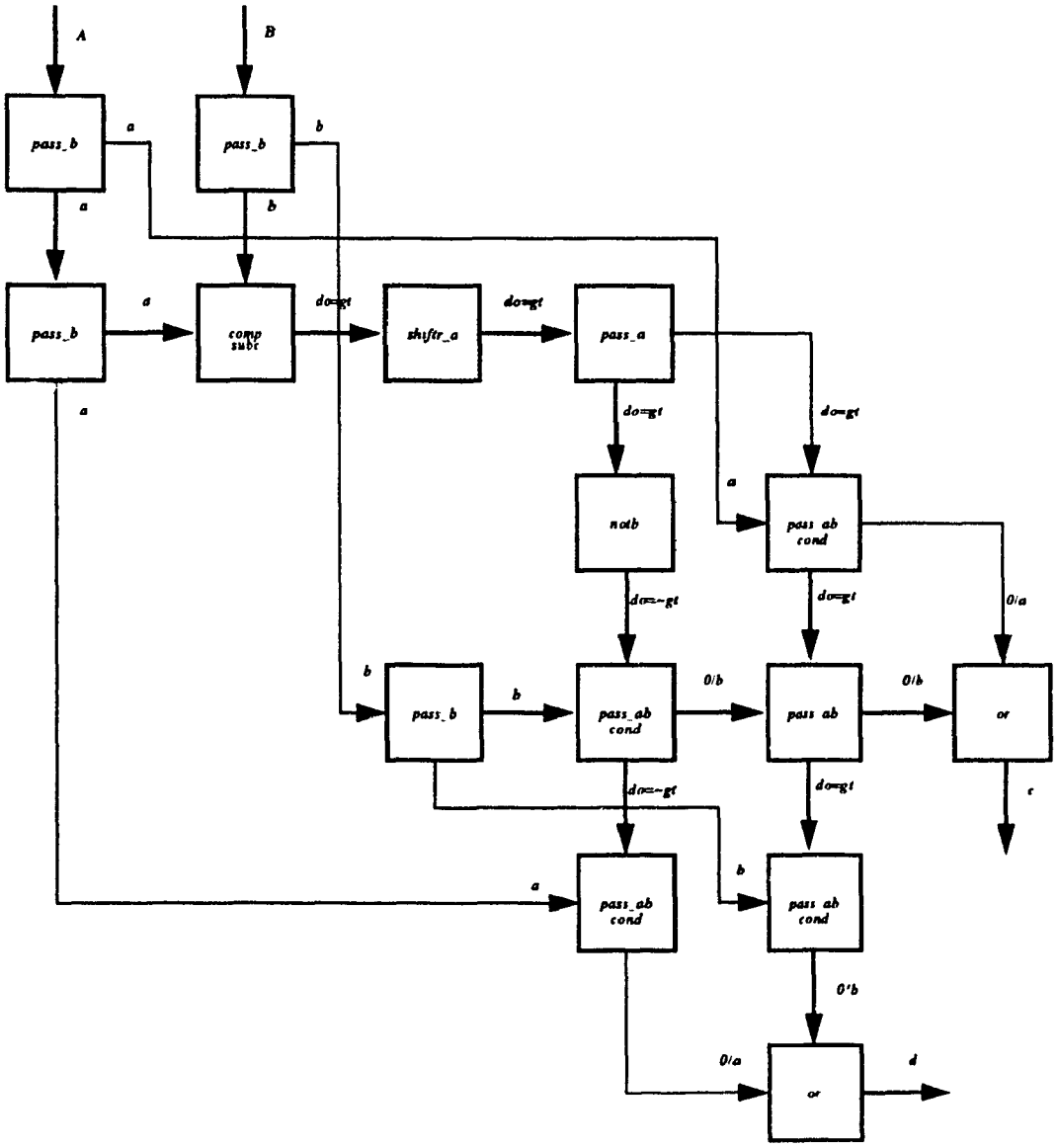


FIGURE 20. A Descending Sort Algorithmic Unit Implementation

The binary sort algorithm can then be implemented as shown in figure 21. The algorithm requires depth -1 phases and partially sorted results must be collected and fed back to subsequent phases.

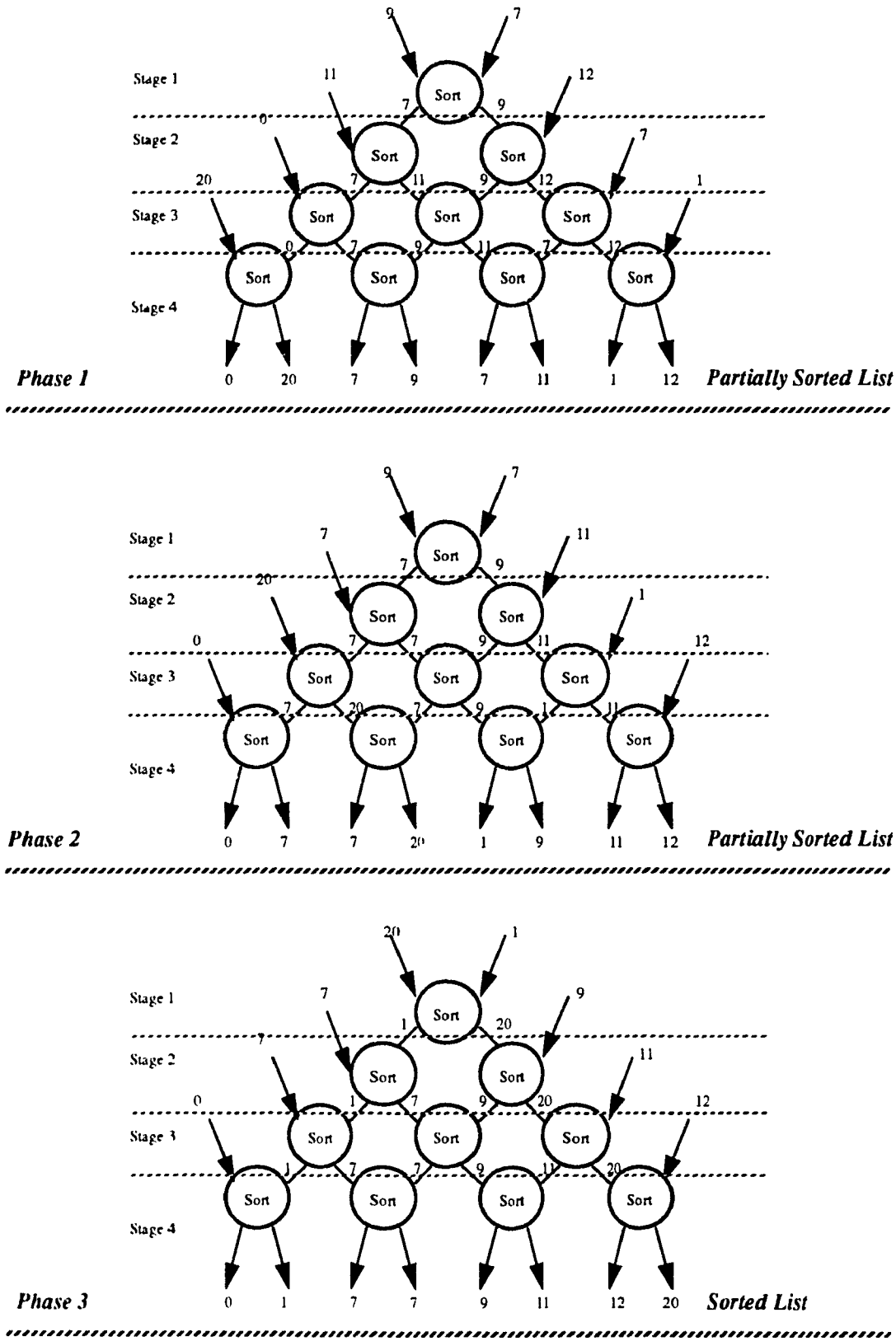


FIGURE 21. An Ascending Sort Algorithm Implementation

4.4.3 A Match Algorithmic Unit

Searching for a match in a list of elements can be implemented as shown in figure 22. This implementation requires that the element index be part of the inputs in order to provide the index as its main output when a match is found. The index may not be zero because the match found mechanism relies on the fact that if a particular match algorithmic unit does not find a match, it will output a zero. If a match is found then the unit's index is placed on the output; however if the index is zero then match found and not found would be the same value, which would be inconsistent.

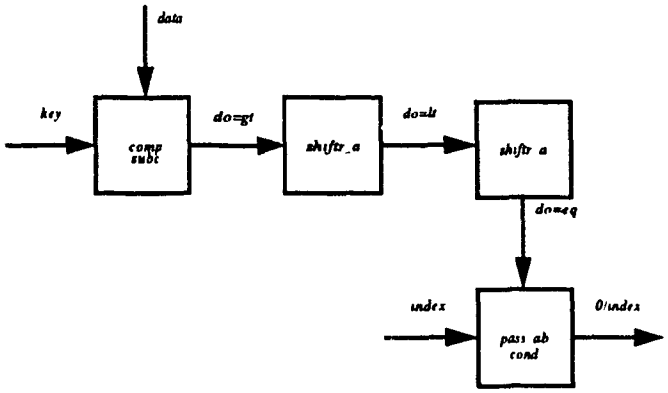


FIGURE 22. A Match Algorithmic Unit Implementation

Searching for a match in a list of elements can then be implemented as shown in figure 23. In this example k is the key we are looking for and d_i is an element of the list we are searching for a match. The input constants are the unit's index. The Gen algorithmic unit is responsible for determining whether or not there is a match. Its implementation can be as simple as an oring of all inputs or as complex as required. The simple implementation would provide the index of the unit that found a match. This implementation assumes that all input list elements are unique because if more than one match is found the resulting index would not be valid.

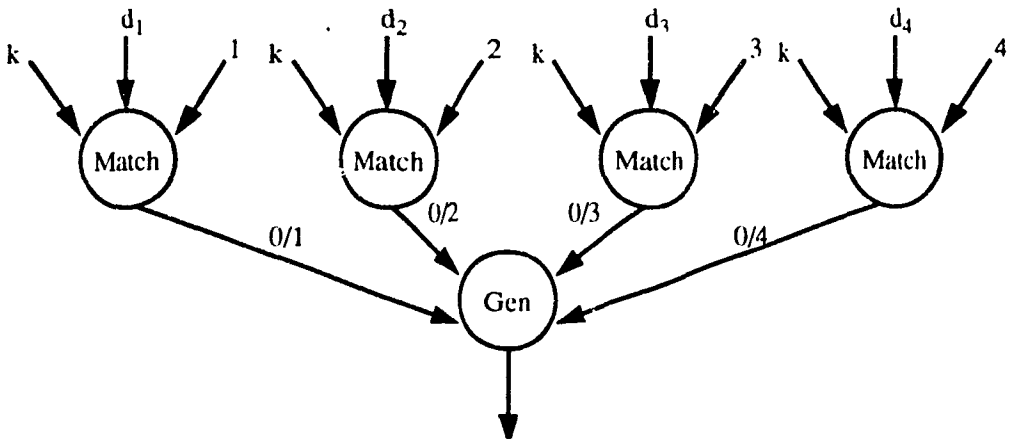


FIGURE 23. A Match Algorithm Implementation

4.4.4 A Factorial Algorithmic Unit

The following factorial algorithm requires a multiply function which the PAAP does not support directly; however, the multiply operation can be built by using simple operations and defined as an algorithmic unit. Let us assume that we have a defined multiply algorithmic unit and attempt to implement the following factorial algorithmic:

```
func factorial(number, result)

    if we have worked on all numbers

    then return with the result

    else factorial(number - 1, result * number)
```

The factorial algorithm implies a recursive mechanism, but the PAAP is incapable of implementing such a mechanism. Instead, the PAAP can be programmed to generate partial results which would be picked up by the host and given right back to a PAAP element for further processing. Such partial results could be considered to be equivalent to one stage of a folded loop or one level of a folded recursion (see figure 24).

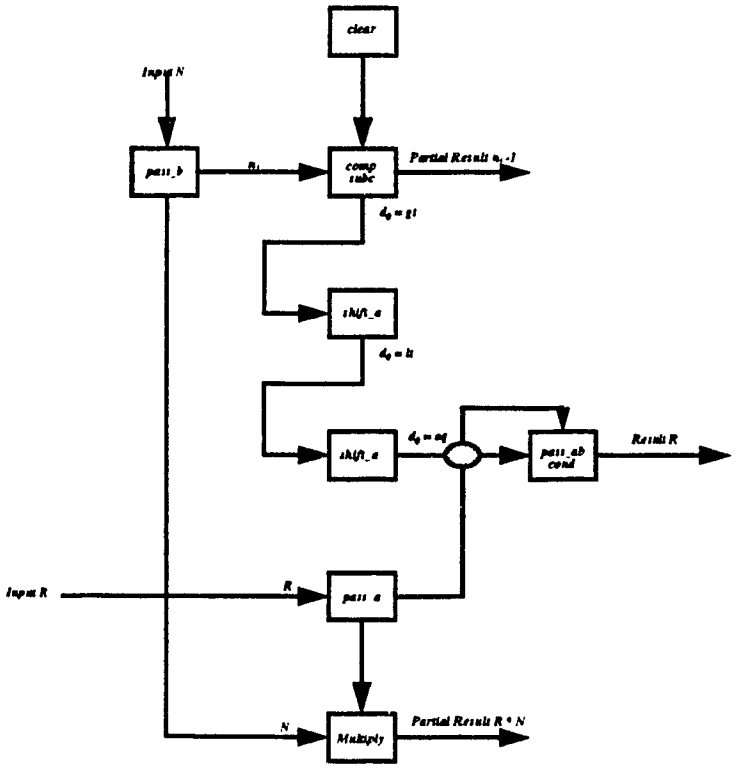


FIGURE 24. A Partial Factorial Program

We must point out that not providing direct looping capabilities or not having a built-in stack for recursion was a conscious decision we had to make in order to better exploit the inherent parallelism in the architecture.

Both N and R are inputs and N - 1, R * N, and R are outputs. The multiply subprogram is denoted by a PE sub-array executing the multiply instruction. The compiler should be able to develop standard implementations for the most-used routines such as multiplication, and use them at will. The stage compares the incoming N value to 0 in order to make sure that the recursion is not at its last stage. If the current stage is not the last, then the incoming N is decreased by 1 and the N-1 partial result is generated. The incoming R is also multiplied by N and the R * N partial result is generated, as well as the full result R is set to 0 because we are not done yet. If the current stage is the last one, then the current incoming R is the result and the result is set to factorial N.

Chapter 5 The PAAP PE and RE Architecture

5.1 A Processing Element

The processing element is the center piece of the design. It is a fully programmable core, with an asynchronous data flow. In figure 25, we can see that the PE is composed of three sub-modules, the sync-input sub-module, which generates the required input timing signal for the PE core to carry out its computation, the PE core sub-module itself, which implements all of the PE's functionality, and the sync-output sub-module, which generates the required output timing signal for the neighboring PEs.

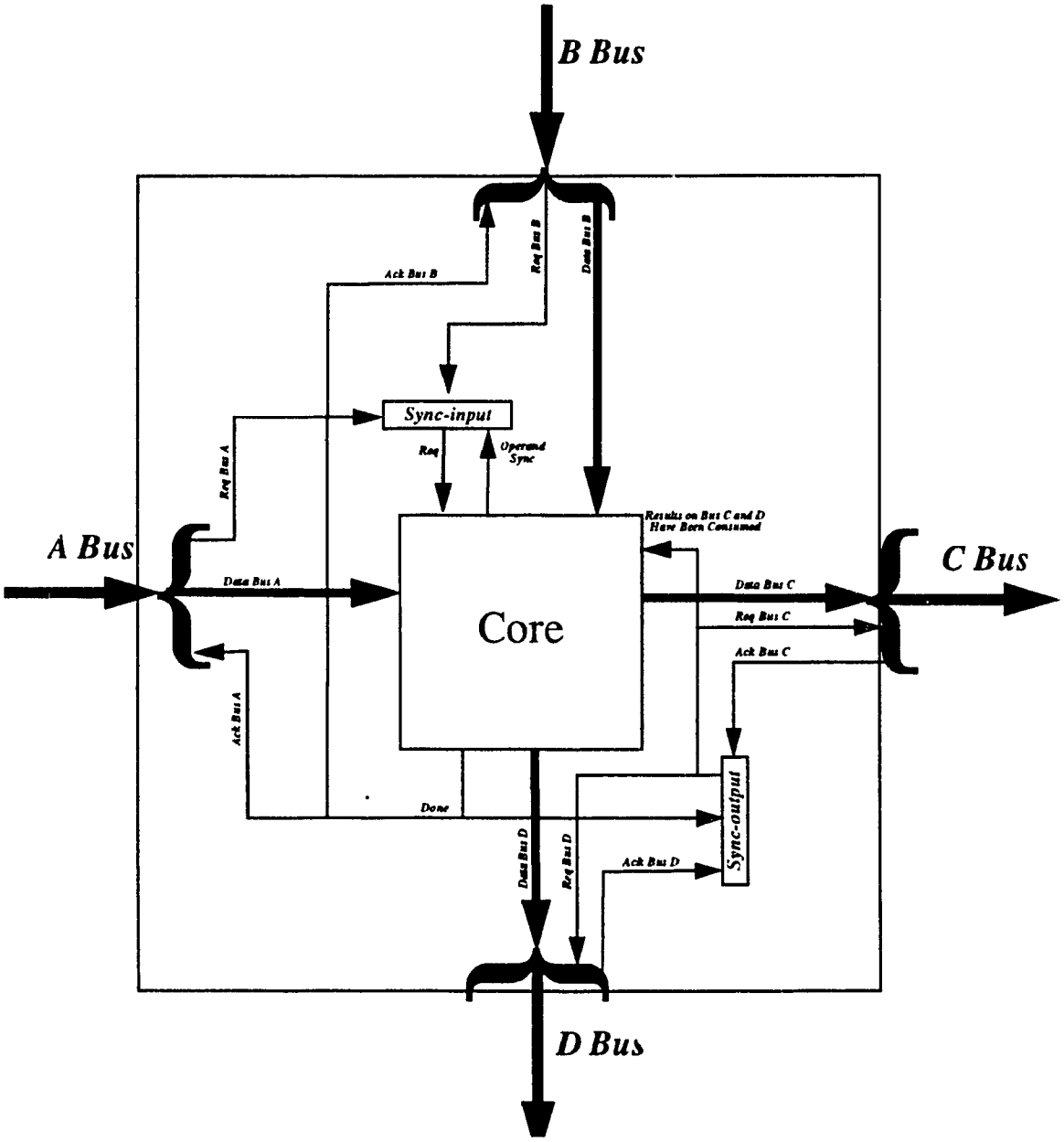


FIGURE 25. The Processing Element Block Diagram

The inter-PE communication is totally asynchronous and follows the timing diagram shown in figure 26. The idea is that when PE(i,j), in a PAAP array of N by N elements, has results ready for consumption by the PEs connected to its output busses, it will generate a request signal (req) for each output partner, to signal its partners that they should consume the results. The PE(i,j) does not necessarily have to connect its output to its right and bottom neighbors. The PE(i,j)'s output partners could be anywhere in the PAAP array where it is possible to make a connection via the programmable on-board REs.

Once the req signal has been generated, PE(i,j) will wait for a signal from both its output partners telling it that they have consumed the given data and that PE(i,j) should go on to its next operation. In reality, PE(i,j) will process its next data set if present, while waiting for its output partners to consume the previous results. However, the new results will not be latched into the results register until PE(i,j) has received the proper signal from its output partners. This allows us to maintain the maximum level of parallelism without compromising the timing constraints. Its implementation is accomplished by having the core's input busses always sample the PE input busses; when new values are present in the core's input busses, the instruction stored in the IR is applied to its inputs and a set of results is generated at the core's outputs, but not latched onto the output registers.

After PE(i,j) has received both ack signals from its output partners, it resets its req signals and goes on to process its next data set. It is possible that PE(i,j) will have a new result ready soon after the consumed signal appears, in which case the cycle will start again. However, if there are no available inputs to process at the time the consumed signal arrives, it will remember that it is ready to go, and as soon as there are any req signals on its inputs it will carry out its instruction on the input data and generate its next set of results.

Inter-PE synchronization can be viewed as a five step process as follows:

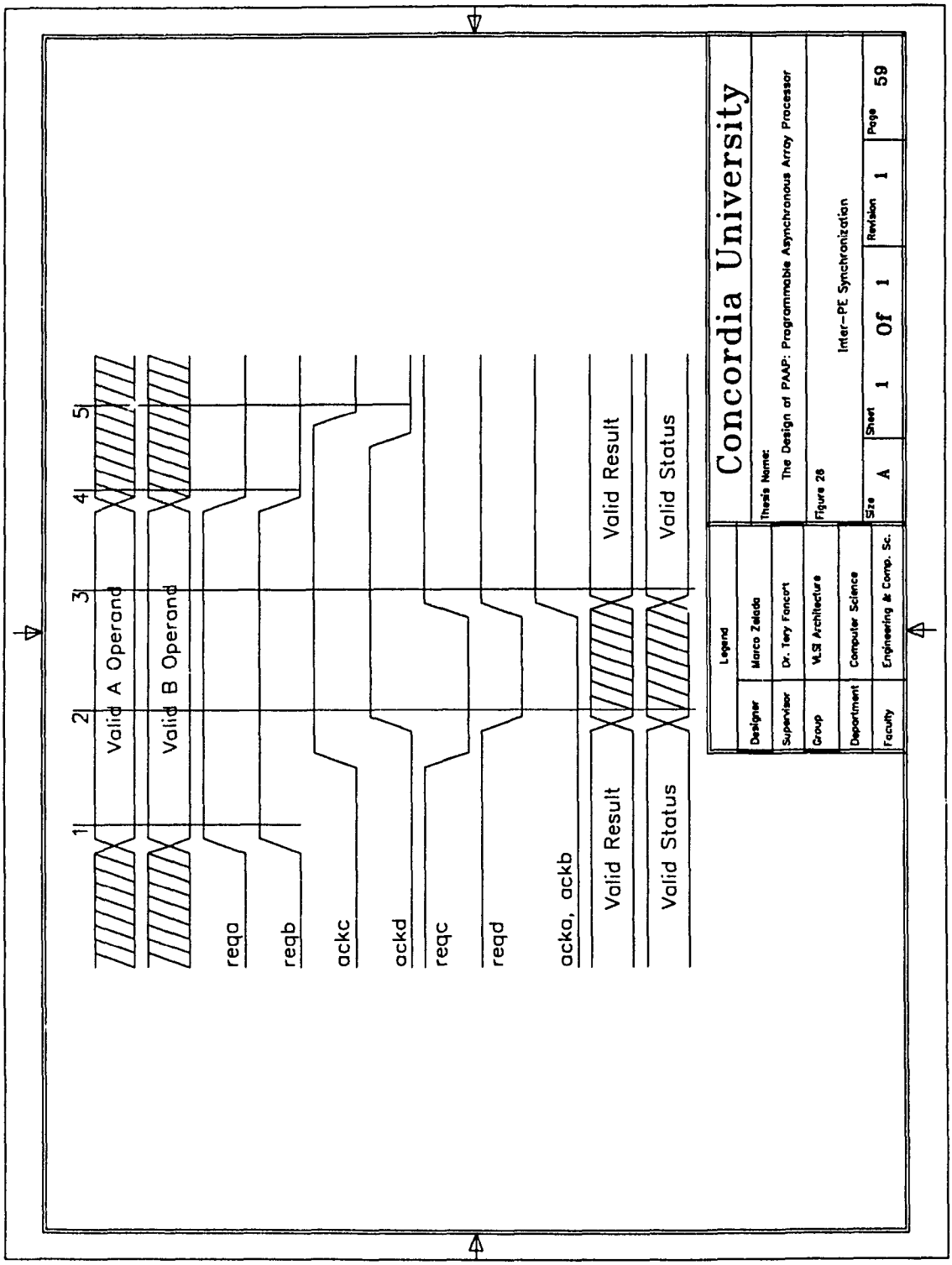
- Input data is valid (Step 1 in figure 26).

  Input data is present in both data busses A and B and it is signaled by the arrival of the reqa and reqb respectively. Computation of the next set of results begins.

  - req^{+}; $\underline{acka}^{+}$;
  - $reqb^{+}$; $\underline{ackb}^{+}$;

- The previous result are consumed (Step 2 in figure 26).

  The previous results are acknowledged by the output partners, which causes the reqc and reqd signals to be reset, reqc and reqd. This begins an output synchronization cycle and is signaled by the arrival of the ackc and ackd synchronization signals.


1
2
3
4
5
Valid A Operand
Valid B Operand
reqa
reqb
ackc
ackd
reqc
reqd
acka, ackb
Valid Result
Valid Status
Valid Result
Valid Status
Concordia University
Legend
Designer
Marco Zaldda
Supervisor
Dr. Tery Fancott
Group
VLSI Architecture
Department
Computer Science
Faculty
Engineering & Comp. Sc.
Thesis Name:
The Design of PAAP: Programmable Asynchronous Array Processor
Figure 28
Inter-PE Synchronization
Size
A
Sheet
1
Of
1
Revision
1
Page
59

- ackc$^+$; reqc$^-$;
- ackd$^+$; reqd$^-$;

- New results are ready (Step 3 in figure 26).

  New results are present on both output busses C and D and their respective request signals are set. The input partners are acknowledged so that they may go on to their next computation.

  - reqc$^+$; ackc$^+$;
  - reqd$^+$; ackd$^+$;

- The input partners reset their reqa and reqb (Step 4 in figure 26).

  The input partners were acknowledged for their data in the previous step. This resets their respective request lines and ends an input synchronization cycle. It implements the return to zero part of the synchronization protocol on the input side.

  The acka and ackb reset the input partner's output req latches. These signals are derived from an internal acknowledge signal that is generated by the PE's core and reset as soon as the syncout block sets its output req latches. In effect, this implements a reset pulse for the input partner's output req latches. [21]

  - acka$^+$; reqa$^-$;
  - ackb$^+$; reqb$^-$;

- The output partners reset their ackc and ackd (Step 5 in figure 26).

  The output partners reset their acknowledge signal once their input request signal is reset. This ends an output synchronization cycle. It implements the return to zero part of the synchronization protocol on the output side.

  - ackc$^-$;
  - ackd$^-$;

This protocol reduces the signaling delay by not requiring that the req and ack signals be returned to zero as part of the protocol. Once the current results are generated and the previous results have been

21. The duration of this pulse is always guaranteed to be larger than the setup and h[illegible]times needed by the input partner's output req latches. Otherwise the protocol may collapse

consumed, the previous request signal is acknowledged. Each req signal is automatically returned to zero as soon as its respective ack signal is received, but this is not a specific timing requirement imposed on the PE synchronization.

The sync-input module schematic diagram and timing diagram are shown in figures 27 and 28 respectively. This module requires inputs from the core to tell it the type of instruction it has stored in its IR, this is related to the available instruction set which includes zero, one and two operand instructions.

The control unit in the PE core generates the following signals for this purpose:

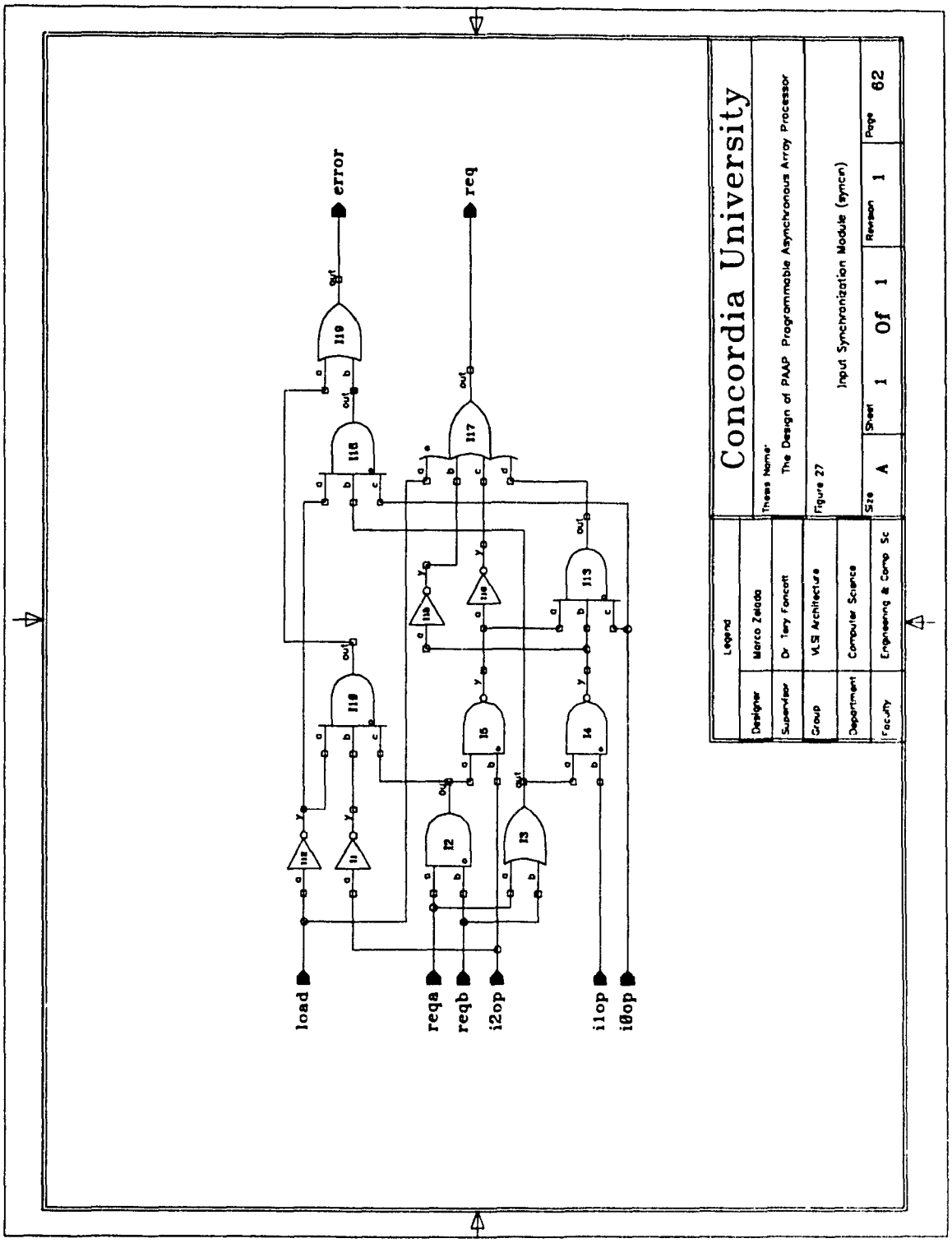
- 0op The instruction in the PE IR has zero operands.
- 1op The instruction in the PE IR has one operands.
- 2op The instruction in the PE IR has two operands.

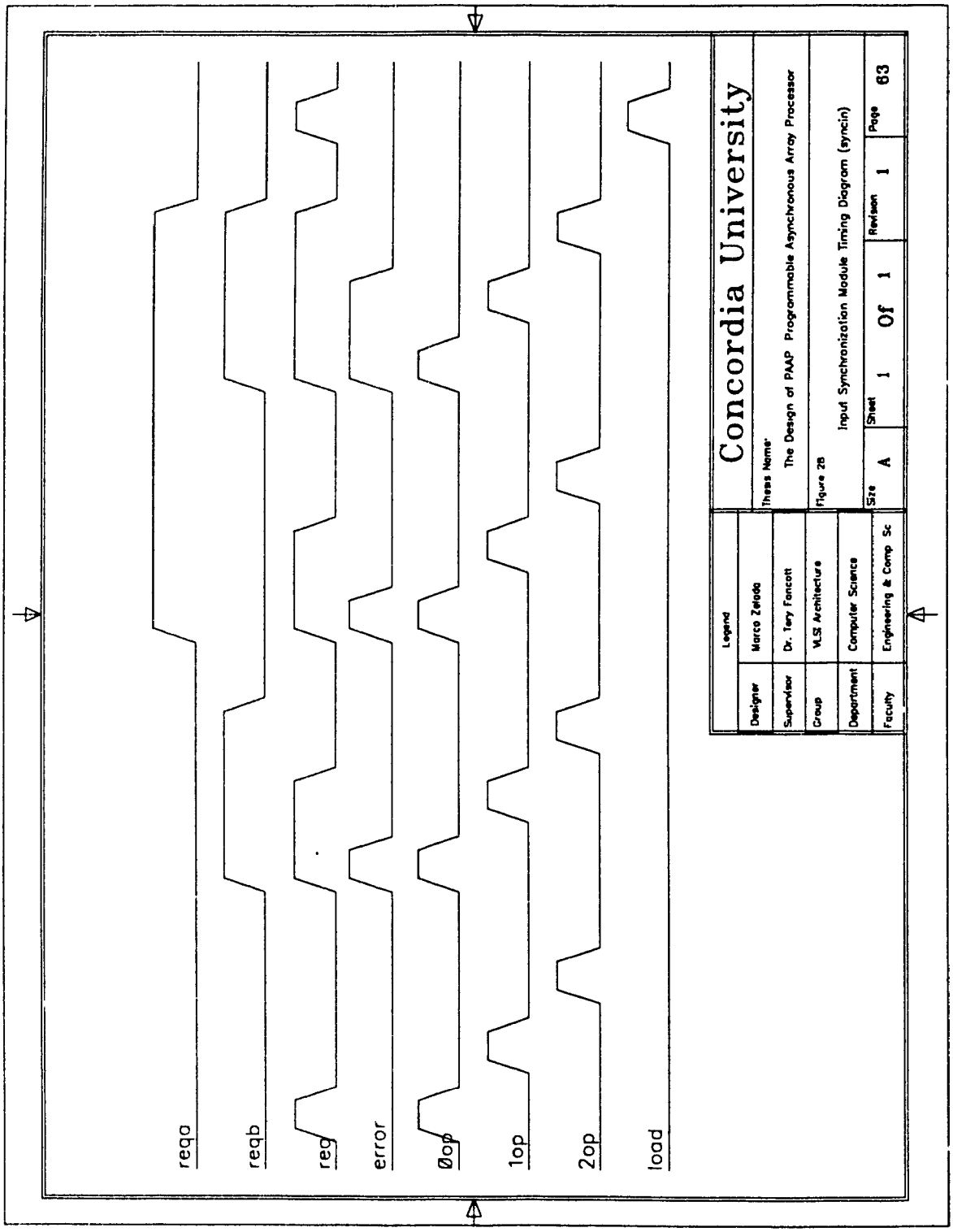
These signals are needed to make sure that the sync-input module can know when to properly generate the core req signal. The sync-input module can not always expect the same number of req signals because this depends on the number of operands that the instruction stored in the IR needs. Such signals also allow the PE to detect when the instruction stored in the IR does not correspond to the number of operands received However in some cases, where the number of received req signals is less that expected, the sync-input module has no way of knowing when the missing req signals will come in.

The PE has no time-out mechanism because it is asynchronous, and it may get into an endless wait cycle due to a problem on its input PEs. There is no easy way to restart a broken req <-> ack chain; it has to be restarted by the host after it realizes that it is not receiving any replies from the a malfunctioning PAAP element. The synchronization circuit is not sensitive to input signal ordering because all input signals are buffered by their driving circuits. Therefore, once a signal is present, it will remain active until the driving circuit changes its value thus avoiding signal ordering problems.

Computational interference is avoided by the use of the consumed signal. This signal becomes active when results have been properly consumed by the output partner PEs. Once a PE has generated its results it must be made to wait for their consumption because otherwise new results could overwrite the previous results This also ensures that the PE core has as few feedback paths as possible which adversely affect the consistency of the stored data in the output registers. The penalty one pays is that 100% concurrent computation, as in systolic arrays, is not possible by all PEs.

The req lines from both busses A and B are taken as inputs to the sync-input module. These signals provide inter-PE synchronization by telling the current PE that it has valid input data on its input busses


error
req
load
reqa
reqb
i2op
i1op
i0op
Concordia University
Legend
Designer
Marco Zelaba
Supervisor
Dr Tony Fancott
Group
VLSI Architecture
Department
Computer Science
Faculty
Engineering & Comp Sc
Thesis Name:
The Design of PAAP Programmable Asynchronous Array Processor
Figure 27
Input Synchronization Module (syncn)
Size
A
Sheet
1
Of
1
Revision
1
Page
62


reqa
reqb
req
error
Øop
1op
2op
load
Concordia University
Legend
Designer
Marco Zegoda
Supervisor
Dr. Terry Fancott
Group
VLSI Architecture
Department
Computer Science
Faculty
Engineering & Comp Sc
Thesis Name:
The Design of PAAP Programmable Asynchronous Array Processor
Figure 28
Input Synchronization Module Timing Diagram (syncin)
Size
A
Sheet
1
Of
1
Revision
1
Page
63

and to go ahead and consume them. The PEs which are providing the reqa and reqb signals along with data on their respective busses will then wait until the current PE signals them that it has consumed the input data. The sync-input module also takes the load signal and uses it to force its output high. This is done to do away with synchronization requirements while the PAAP is being programmed.

The sync-input module may generate a synchronization error in the following situations:

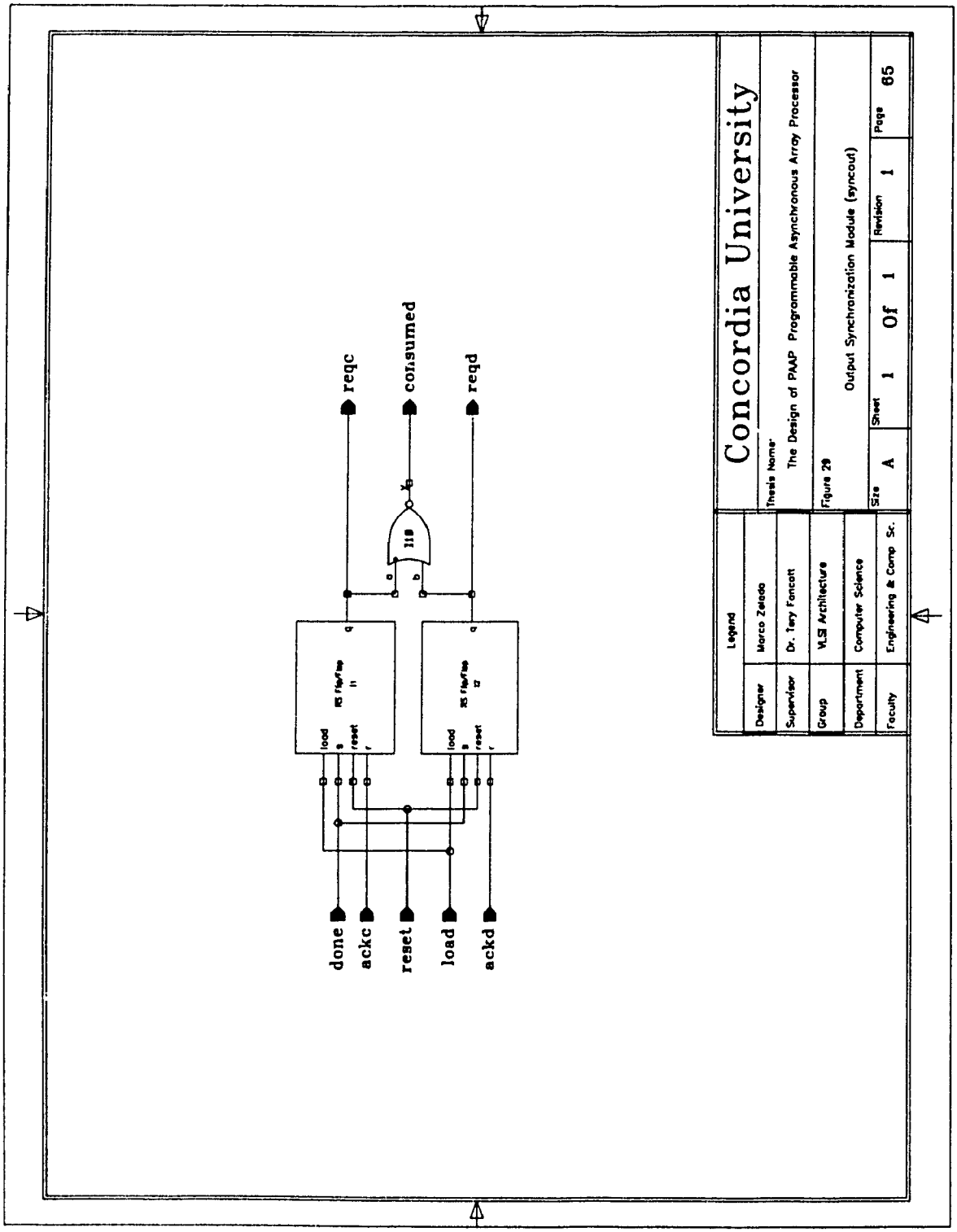
- We have both reqa and reqb present and the CU does not signal a two-operand instruction.
- We have at least one input req signal, either reqa or reqb, and the CU signals a zero-operand instruction.

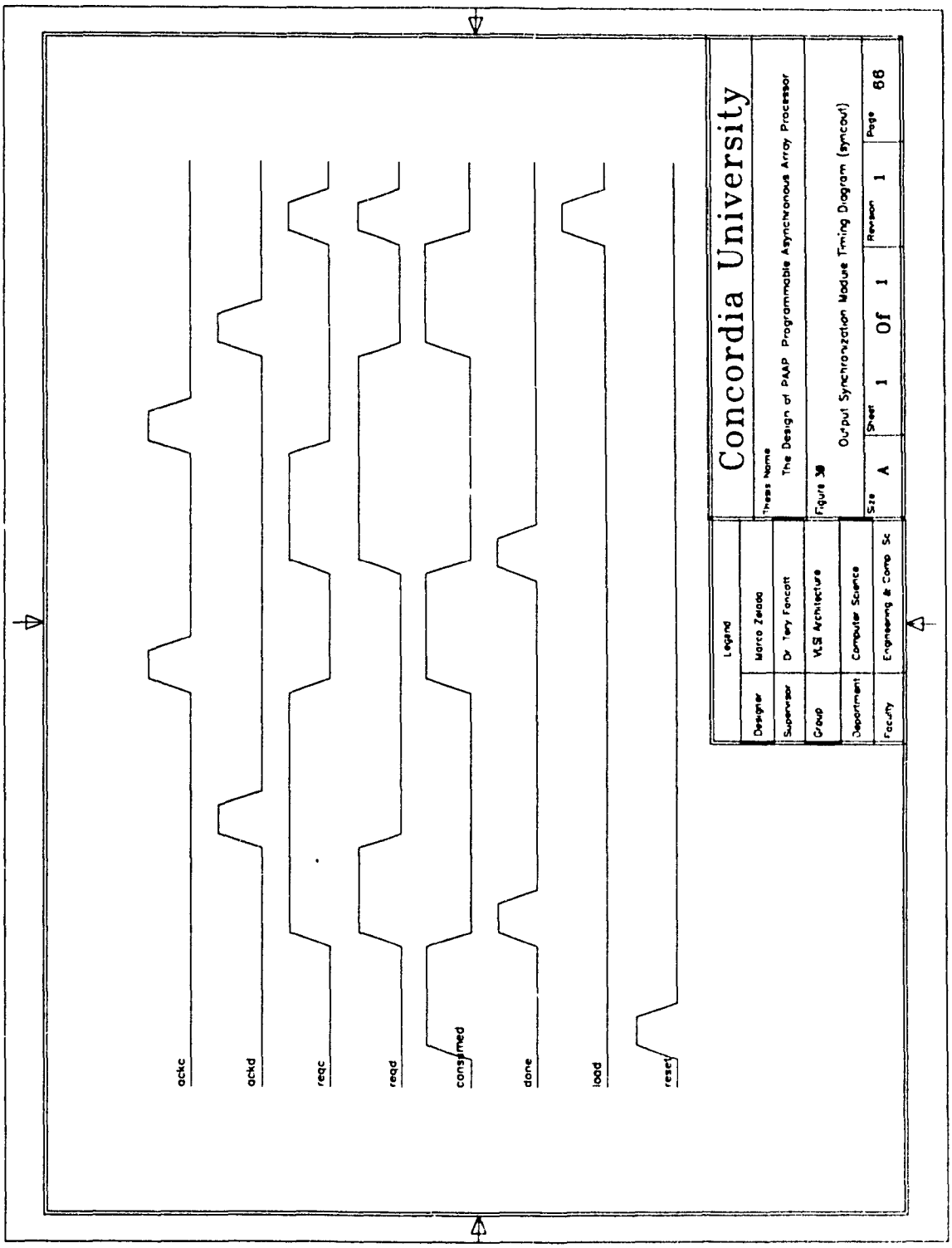
If the CU signals a one- or two-operand instruction and only one or no req is present, we do not signal an error because it is possible that the remaining req signal will arrive later on. All it means is that the current PE has to wait as long as it takes for the required req to arrive. The sync-input module may detect a synchronization error and still produce its req signal. This is to ensure a continuous data flow, but the synchronization error will be included in the core's status word to make it available for other PEs to examine or for the host to use. In the case that two operands arrive and only one is expected, then only one of them will be used and the other will be ignored. The choice of which operand is ignore depends on the one operand instruction being executed.

The sync-output module schematic diagram and timing diagram are shown in figures 29 and 30 respectively. This module is simple in its operation; it takes both ack signals coming from the C and D busses as its main inputs and keeps track of when such signals arrived, so that it can know when both ack signals have been received. These ack signals also serve to reset the req signals that were placed on their respective busses along with valid data. The consumed signal meant for the PE core is generated when both req signals have been cleared.

The cycle is set in motion by the done signal generated by the PE core; this sets up the req signals on the C and D busses. The SR flip-flops used in this design are special in that they take the load signal into account and automatically set their output high when the load signal is high; the S=R=1 input combination sets the output Q=1 instead of setting it to the unknown state. We have been very careful to design both synchronization units to make sure that they will work properly when the PAAP is being programmed. It is crucial that these synchronization units become transparent when loading the PAAP because loading is done columns at a time and this synchronization scheme does not allow such a scheme.

The sync-output sub-module makes sure that the core sub-module does not execute another instruction until the previously generated results have been consumed.


reqc
consumed
reqd
done
ackc
reset
load
ackd
load
reset
Concordia University
Legend
Designer
Marco Zelado
Supervisor
Dr. Tery Fancett
Group
VLSI Architecture
Department
Computer Science
Faculty
Engineering & Comp Sc.
Thesis Name:
The Design of PAAP Programmable Asynchronous Array Processor
Figure 29
Output Synchronization Module (syncout)
Size
A
Sheet
1
Of
1
Revision
1
Page
65


ackc
ackd
reqc
reqd
consumed
done
load
reset
Concordia University
Legend
Designer
Marco Zaddo
Supervisor
Dr Tony Fancott
Group
VLSI Architecture
Department
Computer Science
Faculty
Engineering & Comp Sc
Thesis Name
The Design of PAAP Programmable Asynchronous Array Processor
Figure 38
Output Synchronization Modules Timing Diagram (syncout)
Size
A
Sheet
1
Of
1
Revision
1
Page
66

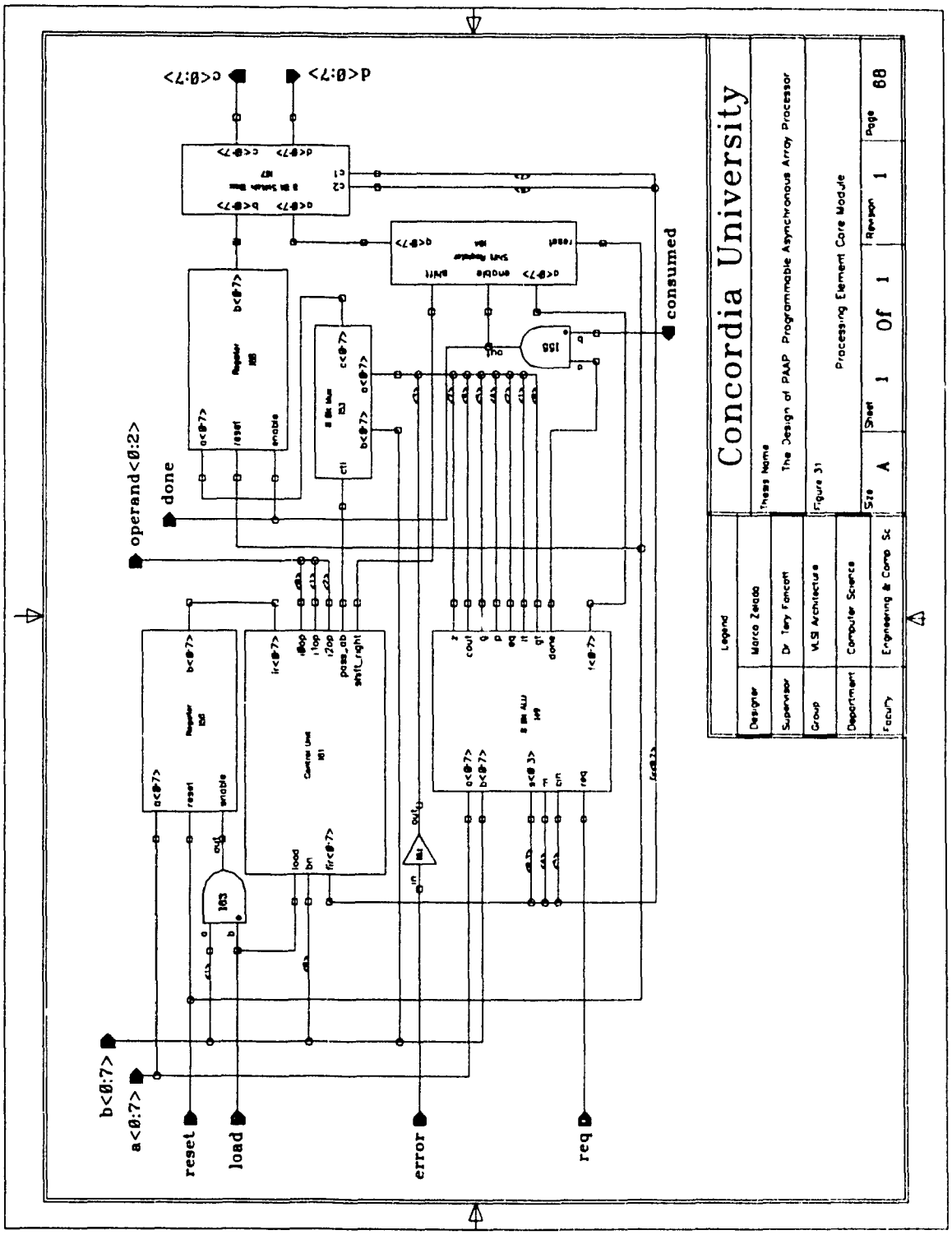
The core begins its computation as soon as it receives the req signal from the sync-input module; however, it requires that the req signal be kept high until it finishes its computation. This is assured by only acknowledging the PEs connected to busses A and B after the core has generated its done signal. The PE input consumed sent on busses A and B are generated by the core done signal. Thus, by the time the input PEs receive an ack signal and reset their req signals, their respective req signals are no longer needed.

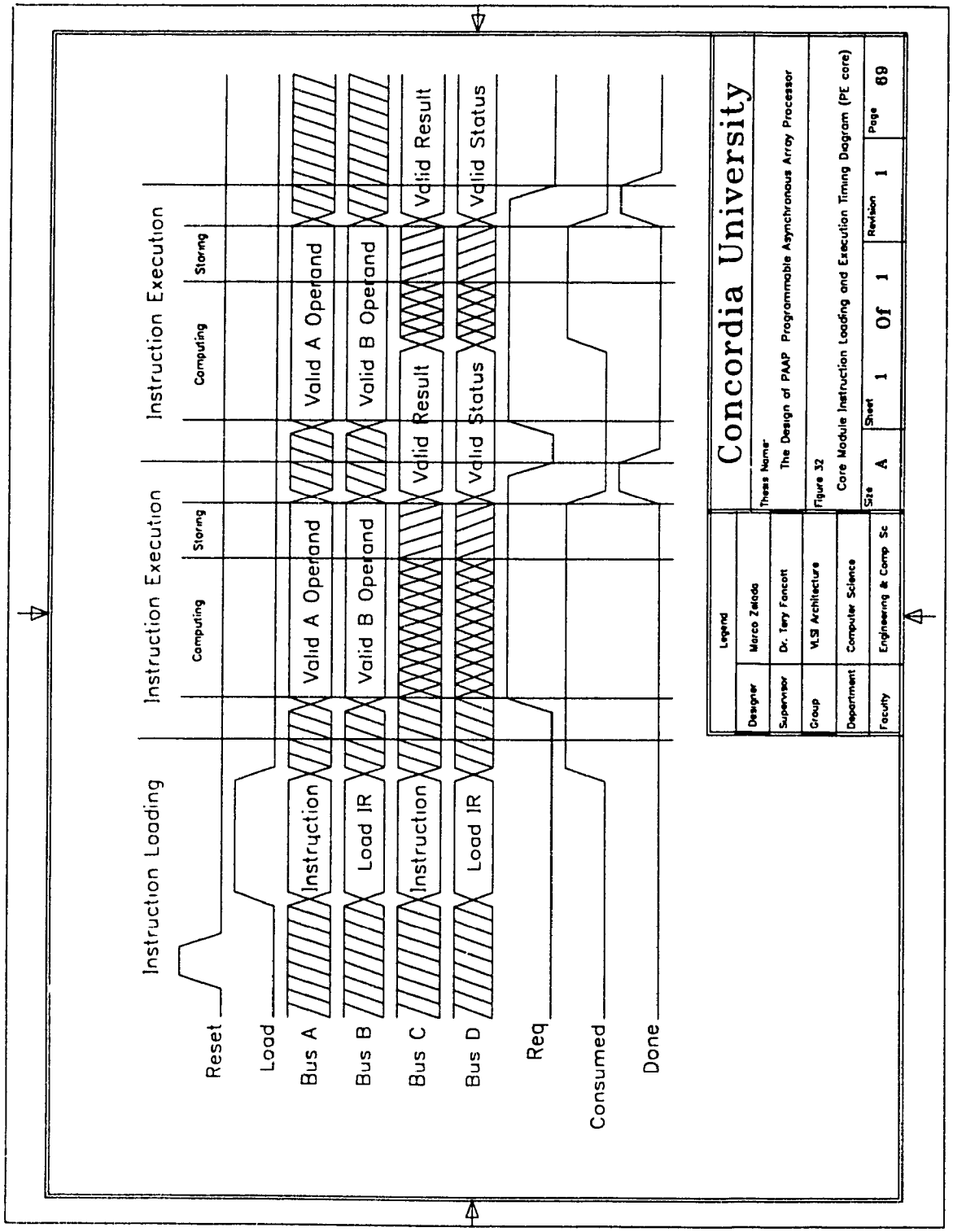
The core schematic and timing diagrams are shown in figures 31 and 32. Instruction loading starts by resetting all latches. All core registers get reset on the falling edge of the reset signal. The instruction to be loaded is placed on the A bus. This is an 8 bit number that uses bits 6-7 to setup the output broadcast configuration, and bits 0-5 to determine the instruction to execute. The hex value 0x02 is set on the B bus to signal that the PE IR is being loaded. Then the load signal is toggled from 1 to 0. The IR is loaded on the falling edge of the load signal. The core instruction execution is quite basic and dictated by the PE I/O synchronization outlined earlier. The core takes its input data from busses A and B which are 8-bit-wide busses. It also requires the load and reset global signals to properly set its operational mode. The req and error signals come from the sync-input module and serve as the means of PE synchronization. The error signals a synchronization error and thus is placed on the status bus of the core so that this information is made available to other PEs and host.

The core outputs 2op, 1op, and 0op are generated by the core's control unit. They are just passed along as core outputs to the sync-input module. The done signal is used to generate ack signals on both input busses, and to generate req signals on both output busses. The output bus C from the core's ALU contains the results of the operation performed by the core on the input data sets. The output data bus D from the core's ALU contains the status of the instruction just executed. Its bits can be interpreted as shown in table 10.

TABLE 10. The ALU Status Bus Bits

ALU Bit	*Purpose*	*Meaning*
D7	z	Result was Zero
D6	cout	Carry out of arithmetic operation
D5	g	The PE G term used for carry look ahead
D4	p	The PE P term used for carry look ahead
D3	error	There was a synchronization error
D2	eq	Operands were arithmetically equal
D1	lt	Operand A was arithmetically < operand B
D0	gt	Operand A was arithmetically > operand B


Concordia University
Thesis Name
The Design of PAAP Programmable Asynchronous Array Processor
Figure 31
Processing Element Core Module
Size A
Sheet 1 Of 1
Revision 1
Page 68
Legend
Designer Marco Zeqaq
Supervisor Dr Tony Fancott
Group VLSI Architecture
Department Computer Science
Faculty Engineering & Comp Sc
operand<0:2>
done
consumed
reset
load
error
req
a<0:7>
b<0:7>
c<0:7>
d<0:7>


Instruction Loading
Instruction Execution
Instruction Execution
Computing
Storing
Computing
Storing
Reset
Load
Bus A
Bus B
Bus C
Bus D
Req
Consumed
Done
Instruction
Load IR
Instruction
Load IR
Valid A Operand
Valid B Operand
Valid Result
Valid Status
Valid A Operand
Valid B Operand
Valid Result
Valid Status
Legend
Designer
Marco Zaleda
Supervisor
Dr. Terry Fancott
Group
VLSI Architecture
Department
Computer Science
Faculty
Engineering & Comp Sc
Concordia University
Thesis Name:
The Design of PAAP Programmable Asynchronous Array Processor
Figure 32
Core Module Instruction Loading and Execution Timing Diagram (PE core)
Size
A
Sheet
1
Of
1
Revision
1
Page
69

D0, D1, and D2 only have meaning when the executed instruction is comp_subc, which means that the ALU will place a full comparison result on the appropriate status bus bits. This instruction also generates A - B - 1 as its result. D7 in this status bus always means that the value of the result bus is zero, this meaning is consistent for all instructions. Bits D6, D5 and D4 only have any meaning during arithmetic operations, and D3 is only present when there has been a synchronization error.

The control unit is so simple that it does not use any instruction decoding for the purposes of execution; however, there is instruction decoding for synchronization purposes. The values stored in the IR are fed into the control unit for several reasons:

- To be able to force instructions on the PE core.The CU is able to force different instructions on the ALU by taking in the loaded instruction bits and properly modifying them to achieve the desired result.

- To implement extra instructions. The ALU does not provide the pass_ab and pass_abcond. The pass_ab mirrors its inputs onto its outputs. The pass_abcond does the same, provided that the bit 0 on the B input bus is high. These instructions are needed to be able to implement "if (condition) then (statement) else (statement)" primitives at the compiler level.

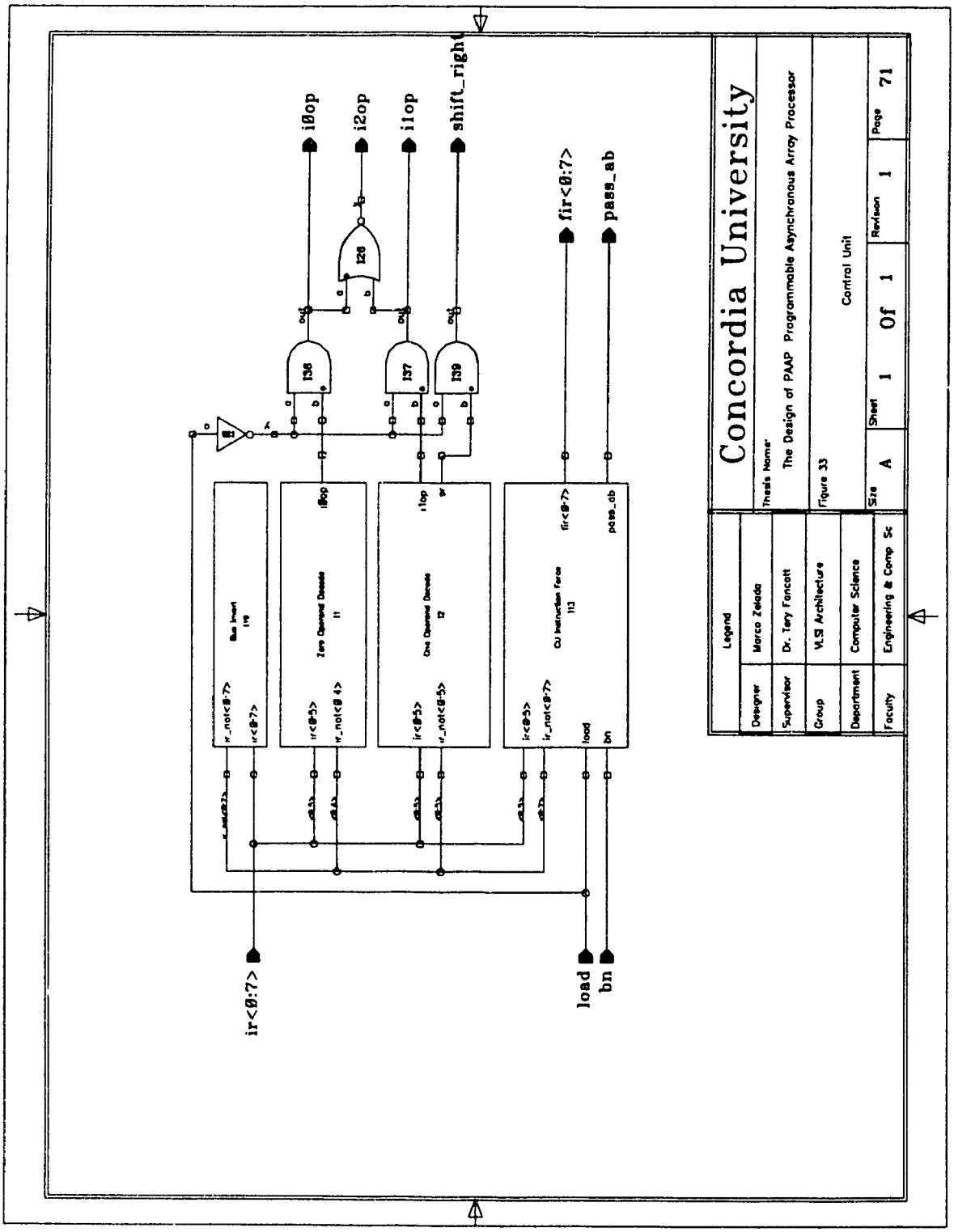
- To decode the stored instruction and signal the sync-input module This is how the sync-input module can be told how many operands to expect (see figure 33 for the CU schematic diagram).

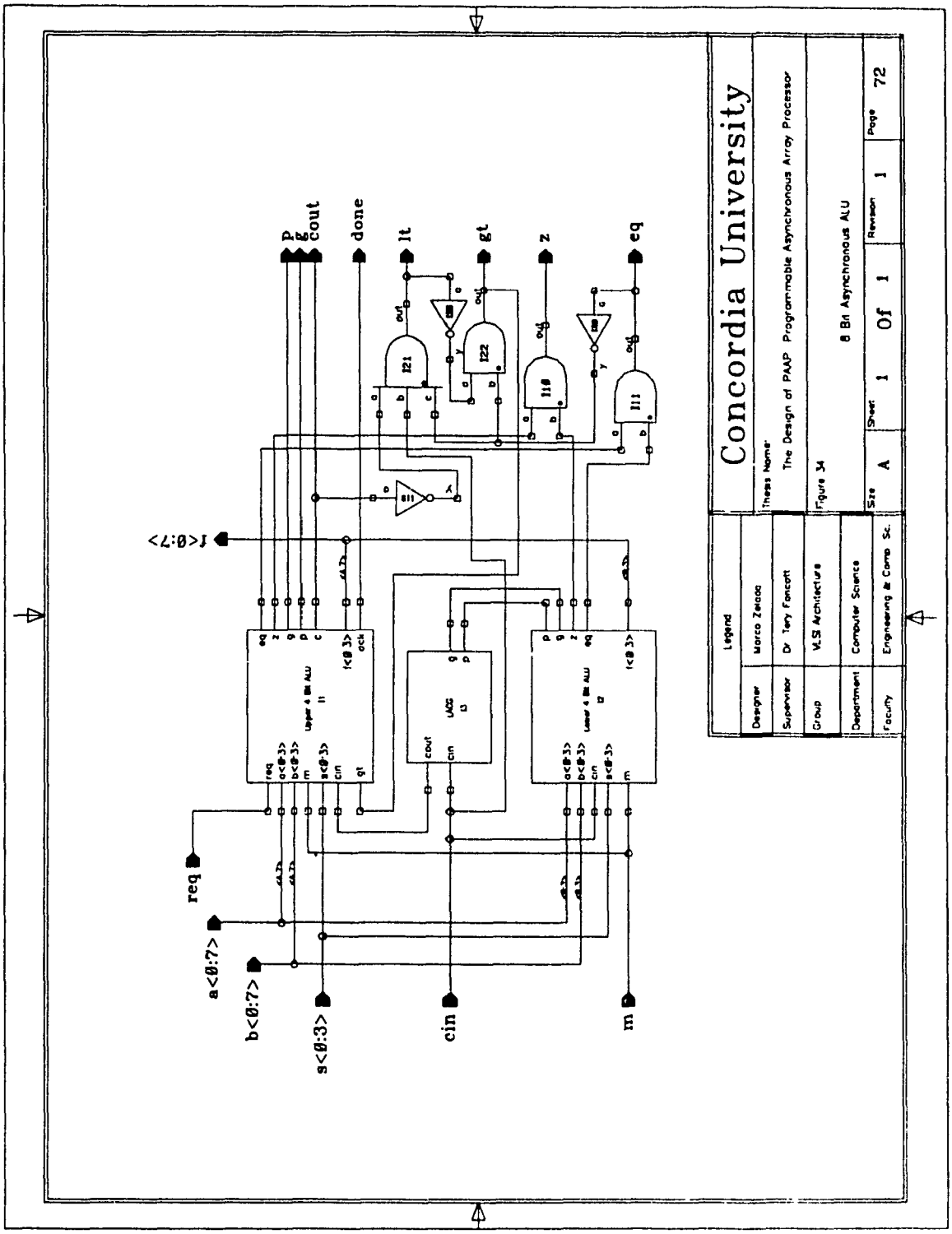
The ALU takes its carry in, mode selection and operation selection bits from the IR. There is no way to have the ALU take its control signals from the input busses. The ALU has internal circuitry that takes the input req signal and uses it to derive an internal core done signal. This internal done signal can be looked at as the longest path through the ALU, so by the time this signal is active, the ALU will have valid results on its output busses (see figure 34). The longest measured delay path through the ALU is the calculation of the ALU's carry out signal. By properly tapping into the input lines used to derive the carry out signal, and using the input req signal to make sure that we account for input decode delays, we can generate a core done signal that guarantees to be the longest path through the ALU.[22]

The ALU is composed of two 4-bit ALU joined with a carry look ahead circuit. The ALU is based on the TI 74181 part. We modified the ALU to take care of the internal done generation and to add all of the necessary status bits. For example, the result = 0 status bit, or the A > B, and A < B status bits were added to the original design [TTLBook85].

---

22 Note that we had to buffer the lower bit signals used to derive the done signal because the extra circuitry exceeded the initially targeted gate fanout


Concordia University
Thesis Name:
The Design of PAAP Programmable Asynchronous Array Processor
Figure 33
Control Unit
Legend
Designer
Marco Zekala
Supervisor
Dr. Tery Fancott
Group
VLSI Architecture
Department
Computer Science
Faculty
Engineering & Comp Sc
Size
A
Sheet
1 Of 1
Revision
1
Page
71
i0op
i2op
i1op
shift_right
fir<0:7>
pass_ab
ir<0:7>
load
bn
I36
I37
I39
I26


Concordia University
Legend
Designer
Marco Zecca
Supervisor
Dr Tony Fencott
Group
VLSI Architecture
Department
Computer Science
Faculty
Engineering & Comp Sc.
Thesis Name:
The Design of PAAP Programmable Asynchronous Array Processor
Figure 34
8 Bit Asynchronous ALU
Size
A
Sheet
1
Of
1
Revision
1
Page
72
p
g
cout
done
lt
gt
z
eq
f<0:7>
req
a<0:7>
b<0:7>
s<0:3>
cin
m

When the pass_ab instruction is requested, the B bus is forced onto the status bus; otherwise the ALU's status bus will contain the status bits generated by the previous instructions. This function is achieved by having an 8-bit multiplexer that has the B bus and the ALU's status bus as its inputs.

The PE ALU can perform 48 instructions. Of the 48, 41 are included in the PE's instruction set, 5 are repeated instructions and thus not used, and 2 repeated instructions were modified to implement extended instruction functionality such as pass_ab, and pass_abcond. The pass_abcond instruction allows us to test for particular conditions before producing a result. The pass_ab instruction lets us implement either a jump or buffering capability. In addition, we also added a shift right instruction in order to support a primitive division algorithm.

For our purposes, the ALU instruction is composed of only bits 0 through 5. In some algorithms, one might find it useful to utilize a switch-box-type of routing on the ALU outputs. After the ALU outputs are latched they are routed to the core output busses depending on the value stored in bits 6 and 7 of the IR (see table 8 for the PE output routing configuration).

We must be aware that each instruction executed by the PE is done totally independently of the values found in the IR bits 7 and 6. The instruction implemented by the PE is purely dependent on the values found in IR bits 0 to 5; the opcode values shown in table 7 are in hex format and represent the values for IR bits 5 to 0 in that order. There is no real no-op instruction in this instruction set; instead, one of the pass instructions should be utilized (pass_a, pass_b, or pass_ab). On reset, the PE is initialized to the inca instruction.

5.2 A Routing Element

The RE does not have any restrictions on any of the busses and any of them can be either an input or an output at any one time, nor does it need any synchronization because it is static in nature. However, the RE I/O busses are not bidirectional, once an RE has been programmed with a particular routing scheme, each one of its 4 busses will be set to being either an input or output bus. In order to change the state of a bus from input to output or vice-versa, the routing element must be reprogrammed with the new desired configuration.

The routing element is what allows the PAAP to have a programmable data flow. The actual circuit size of an RE is 33% smaller than the circuit size for a PE. [23] We have implemented a very efficient routing mechanism and thus the area is well-utilized.

---

23 From simulation, we have been able to calculate that a PE which includes the core, and both sync-input and sync-output modules, can be implemented using approximately 2934 devices, and an RE can be implemented using 1890 devices

The RE executes its instruction in a static mode; there is no need for any synchronization on the RE or instruction decoding. The RE schematic diagram and timing diagram are shown in figures 35 and 36 respectively. The RE instruction loading gets done in much the same way as the PE IR instruction loading, the only difference is that the B bus has the 0x04 value instead of the 0x02 for the PE IR loading. We decided to use different instruction loading values on the B bus, in order to allow for separate PE and RE programming. This permits the host to dynamically reconfigure data flow without disturbing the PE programs. [24]

The RE IR is 8-bits-wide, which provides 2^8 possible routing combinations, but for our purposes only 12 are allowed (see table 9). The reason for such low number of routing configurations is that each bit in the RE IR contains positional data that controls a routing bit's flow. Therefore, only those RE instructions shown in table 9 must be used, otherwise, unpredictable routing results will be obtained. It behaves much like a large switching element that can be programmed to twelve different configurations. It is left up to the compiler to properly program the RE.

The RE uses one bit-route element per routed-bit. Since each bus is 8-bit-wide and each bus also carries an ack and req signal then the RE needs 10 such elements to properly route all data bits. The settings stored in the RE's IR are used to properly configure each bit-route element. The configuration placed in the RE IR applies to all 10 bits on each one of the I/O busses. There is no way to configure ranges of bits instead of configuring the complete bus. The values in the RE IR are ignored when the load signal is high; the RE is by default placed on the a->c, b->d I/O configuration.

The bit-route circuit is where the actual switching takes place. Its schematic diagram in shown in figure 37. The idea behind this switch is to provide a path from any bus to any other bus. The functionality can be explained as follows:

- Inputs coming from the PEs are collected and switched by the demultiplexers

  Depending on which path needs to be created from an input to an output, the appropriate demux gets its control signals and sets itself to pass its input to one of three outputs.

- New outputs are generated from the merged inputs

---

24 Because PE and RE IR loading gets done with different settings, the PE IR values are not disturbed while the RE IR values are being loaded. The PE is forced into the pass_ab instruction, which makes the PE appear as a buffer on the input busses. This helps in row and column programming. The same applies to dynamically reconfigure PEs, because the RE is automatically configured into the a to c, b to d routing when the load signal is present.

The demux's output is then ored with all other demux outputs to form a single set of outputs. The direction control circuit then isolates those busses configured as outputs, which creates a set of two inputs and two outputs.

Implementing the "a->c, b->d" instruction would be done by setting the bit_route bus to "0xEE", which would set demux4(1) to pass "a" to "e", demux4(2) to pass "a" to "f", demux4(3) to be off, and demux4(4) to be off. All demux4(1-4) outputs that are not selected generate a 0, as well as all outputs on a demux that is turned off. This means that the output of the OR gates I11 and I12 would only have logic values present on input "a" or "b" respectively. The outputs of OR gates I13 and I14 do not affect the inputs "a" and "b" because the direction flow circuit has tri-stated them. The I27 allows I11 to drive the output bus c, thus implementing the "a->c" connection, and I26 allows I12 to drive the output bus d, thus implementing the "b->d" connection.

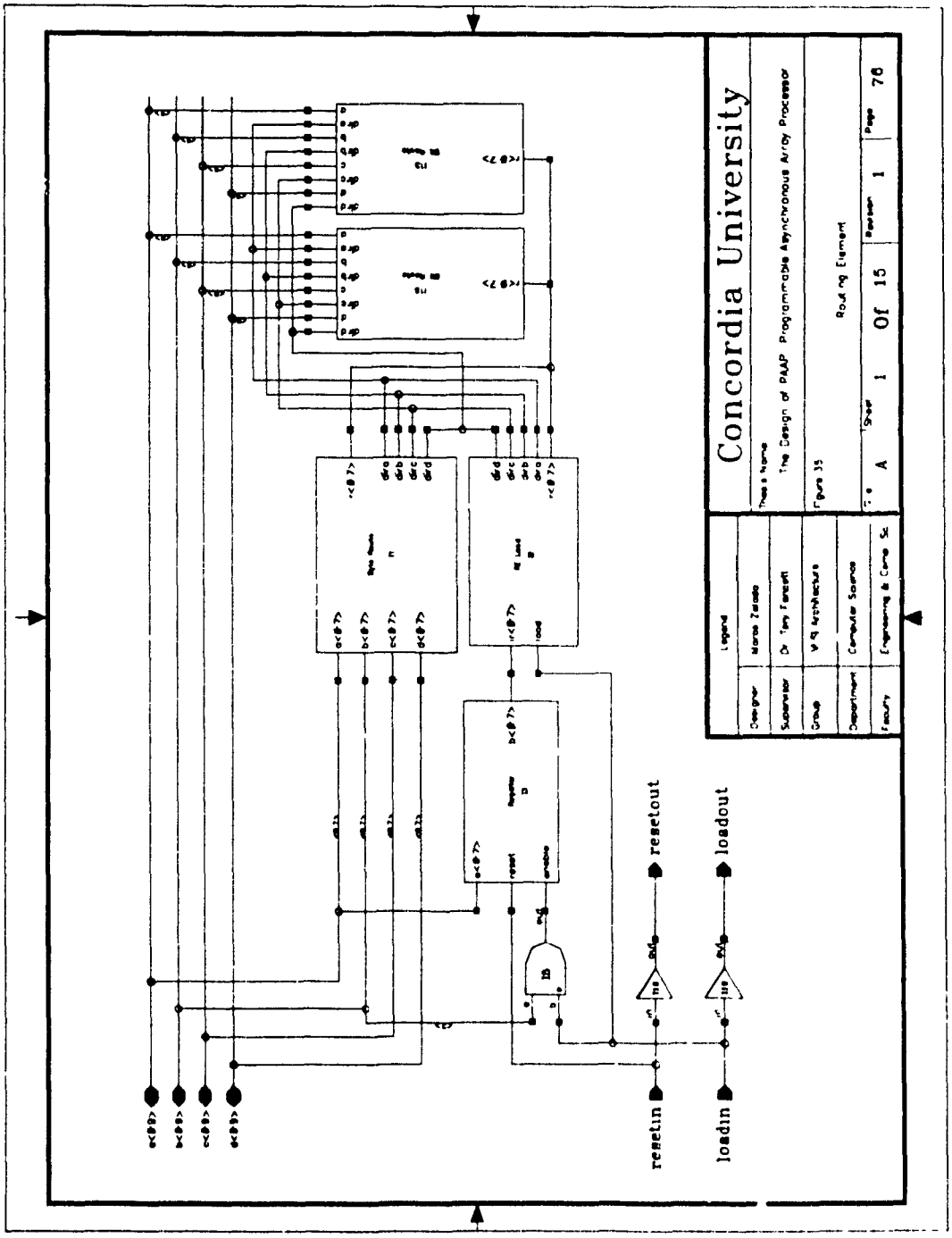
Direction flow on each bit is controlled by a pair transmission gates and bit direction signal provided by the RE logic. This direction signal is derived from the different RE IR bits as follows:

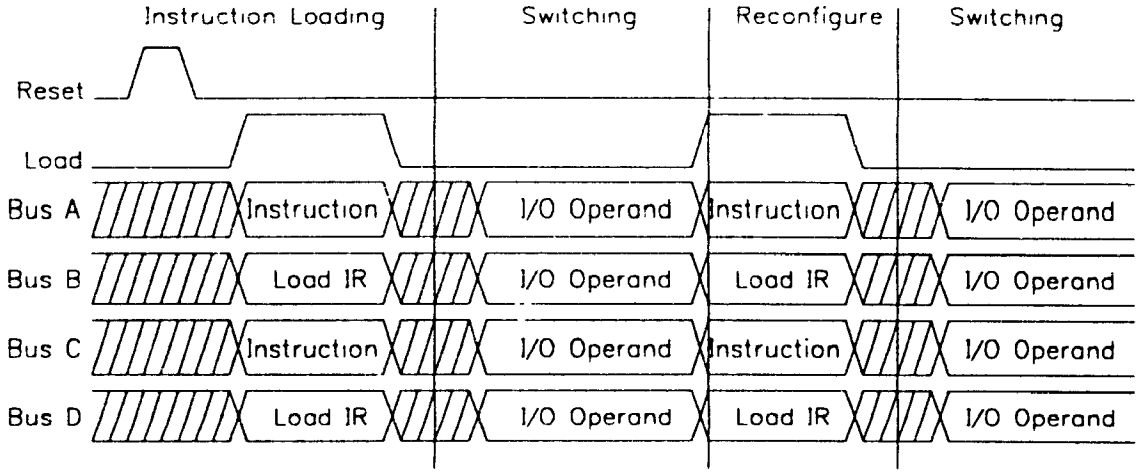
- ir1 = 0, ir0 = 0 then A is an output
- ir3 = 0, ir2 = 1 then B is an output
- ir5 = 1, ir4 = 0 then C is an output
- ir7 = 1, ir6 = 1 then D is an output

When a bit is an output, it must be driven by the RE. However, if the bit is an input then a PE is driving it and the RE must tristate it.

We found that this particular circuit breaks most switch level simulators because of charge sharing or node storage limitations they have. Almost all switch level simulators we tried on this circuit failed to produce the right results initially. However, after researching the particular simulator's documentation, we always found ways to get around their limitations and thus got the proper simulation results.

Our functional simulation of both PE and RE showed us that they work well at around 6.67 MHz. This value was obtained by changing the system clock period in the logic simulators until the circuit broke. The PE signal propagation delay is limited by the longest path defined as the path that generates the synchronization signal and has an upper bound of 80 delay units. The RE signal propagation delay has an upper bound of 10 delay units. A delay units is used here as a measure of delay time for comparison purposes.


Concordia University
The Design of PAAP Programmable Asynchronous Array Processor
Routing Element
resetin
loadin
resetout
loadout



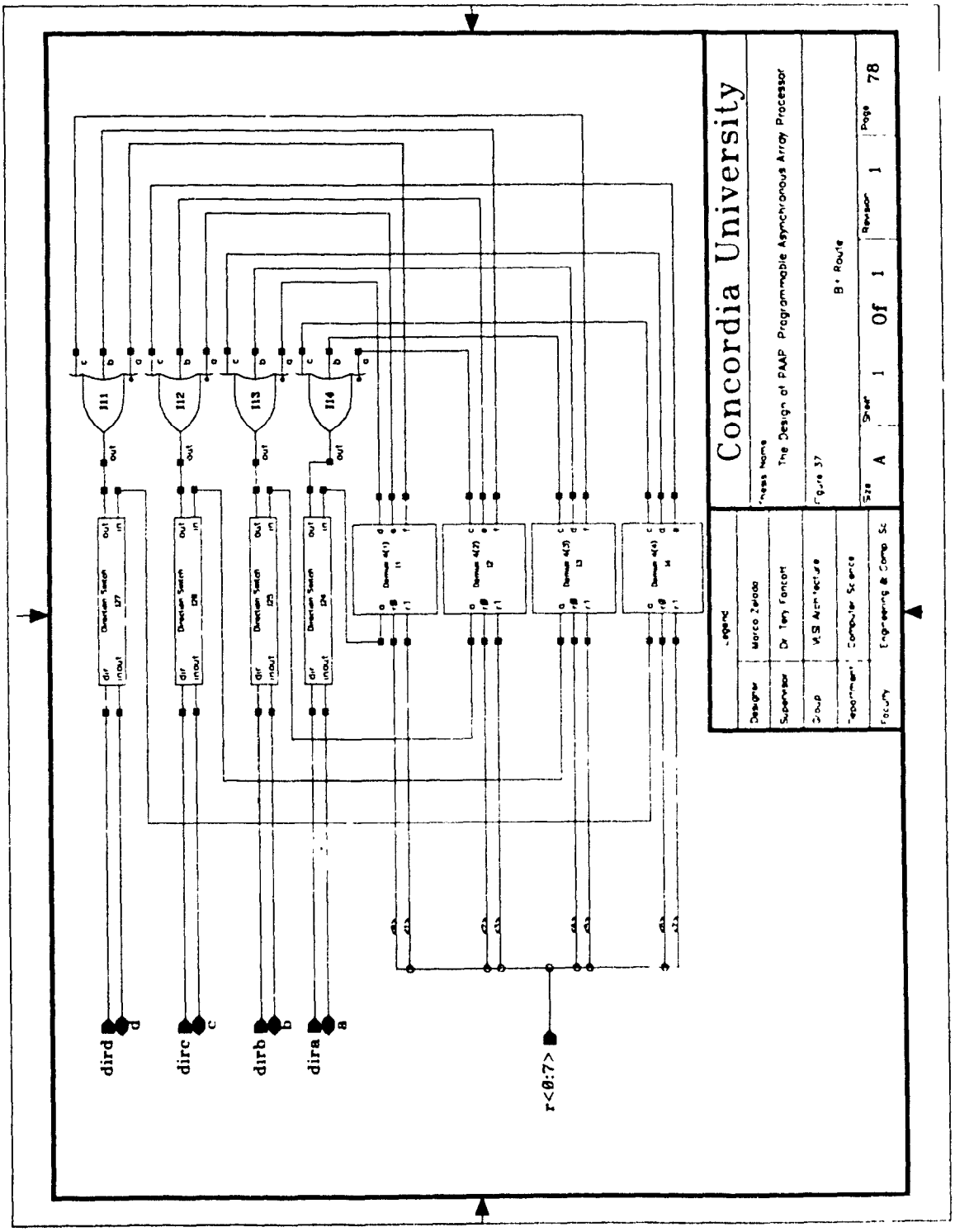
Legend	
Designer	Marco Zaloda
Supervisor	Dr Tery Fancott
Group	VLSI Architecture
Department	Computer Science
Faculty	Engineering & Comp Sc

Concordia University

Thesis Name
The Design of PAAP Programmable Asynchronous Array Processor

Figure 36
RE Instruction Loading and Execution Timing Diagram

Size	Sheet			Revison	Page
A	1	Of	1	1	77


Concordia University
The Design of PAAP Programmable Asynchronous Array Processor
B+ Route
Figure 37
Page 78
dird
dirc
dirb
dira
r<0:7>
I11
I12
I13
I14
Demux 4(1) 11
Demux 4(2) 12
Demux 4(3) 13
Demux 4(4) 14

Chapter 6 Practical Issues

6.1 The Design Environment

The design environment for this project has remained constant even though the computing environment changed dramatically. We started to do the logic design and RNL simulation while still at Concordia University. RNL is an event-driven timing logic simulator for digital MOS circuits, which uses a simple RC model to estimate node transition times and the effect of charge-sharing. The user interface is a simple LISP interpreter, which allows for the programming of complex simulations [RNL83].

As the design process progressed, we implemented minor RNL source code modifications to improve its performance and output formatting capabilities. At the same time, we decided to develop an automated way for stimulus vector generation. The implementation consists of a number of C language header files that describe the desired input stimulus to RNL. Input stimulus generation is thus reduced to writing a C program that describes the Device-Under-Test's (DUT) behavior. This allowed us to generate very complex sets of test patterns that can be retargeted to different simulators. Each simulator would require its own specialized header file to account for its specific I/O format, but most of the header files are common to all simulators for they describe the simulation environment. The same stimulus program is used to drive different simulators; once it is compiled for a particular simulator the executable is used to feed the input stimulus to the simulator via a UNIX pipeline. The C program also provides excellent documentation of the DUT's behavior, and more importantly it could be used to develop IMS-tester test programs to characterize the prototype IC when it is built (see appendix b). [25]

Later in the design process, we obtained the Oct-Tools from UCB and discovered its schematic entry capabilities. The Oct-Tools is a collection of programs and libraries which together form an integrated system for IC design. The system includes tools for PLA and multiple-level logic synthesis, standard-cell placement and routing, and custom-cell design, and a variety of utility programs for manipulating schematic, symbolic, and geometric design data. Most tools are integrated with the Oct data manager and the vem user interface (see figure 38) [HarrisonD86].

We used the RNL circuit descriptions as a documentation aid and entered the design into the vem schematic editor so that we could later use circuit synthesis techniques on it (see appendix c). The Oct-Tools has its own logic simulator called MUSA. It is a multilevel simulator, whose primary mode of operation is switch-level simulation of MOS transistors. The key feature in MUSA was that it makes direct provisions to allow the simulation of higher level constructs such as logic gates, latches, and memory cells;

---

25 An IMS specific header file would have to be defined which maps the C macro definitions into proper IMS commands

it also reads the OCT schematic database as one of its input netlists. However, MUSA treats every node as a storage node. [26]

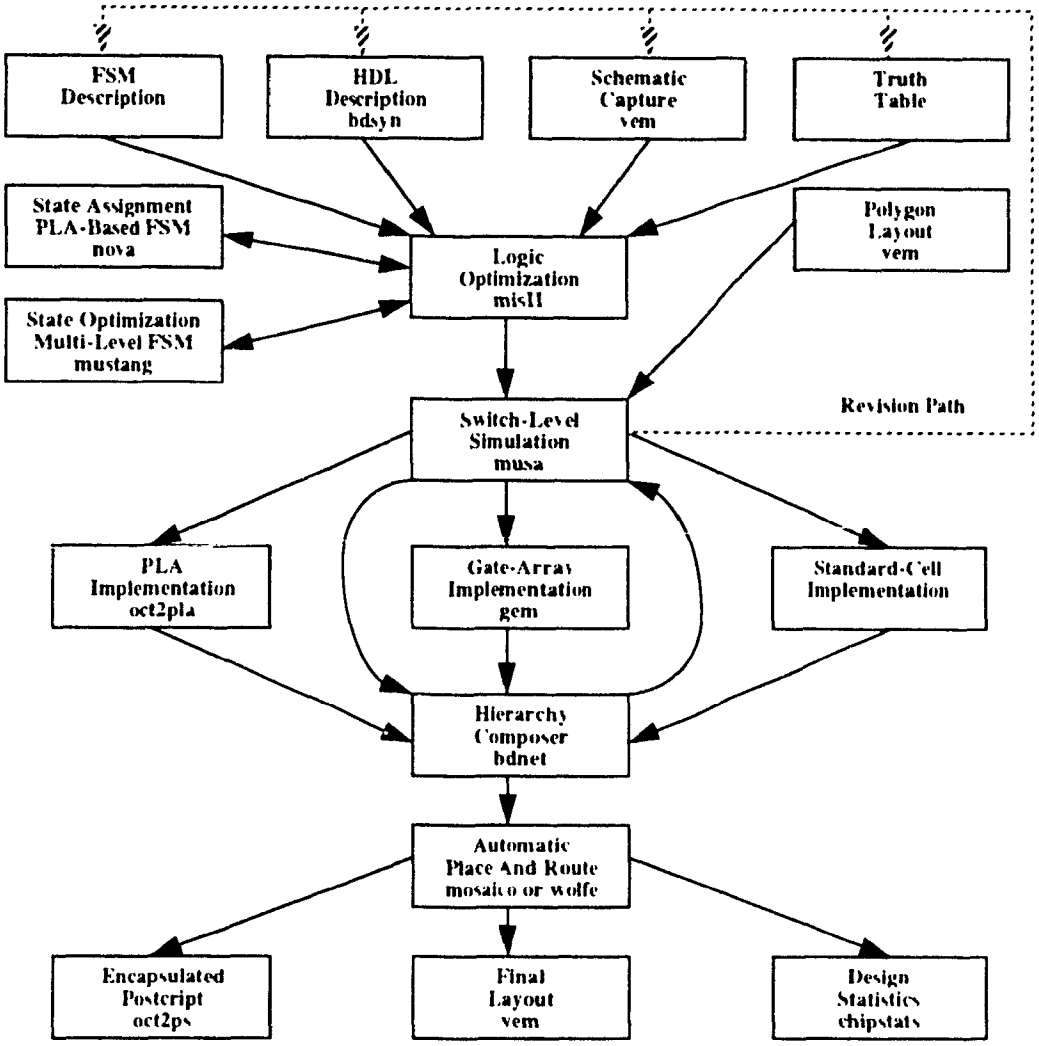


FIGURE 38. The Oct Tool Set Flow Chart

Our strategy at this point was to utilize both RNL and MUSA as a mean of validating the design. This approach helped us in catching a signal strength problem with MUSA, which was not detected by RNL.

The simulated circuits could then be synthesized and mapped onto a particular standard cell library. We could then use the misII tool to automatically generate the layout from the already entered schematics. MisII is an algorithmic multilevel logic synthesis and minimization program. It starts from a description of a combinational logic macrocell and produces an optimized set of logic equations which preserves the I/O behavior of the macrocell. The program includes algorithms for minimizing the area required to implement

26 Once a node is charged high or low, it will remain that way until the node is driven by an instance or set by the user. It is important to note that MUSA does not model charge sharing.

the logic equations, and a technology-mapping step to map a network into a user-specified cell library. This allowed us to concentrate on architectural issues rather than circuit issues, and would help in obtaining the smallest possible layout. Since we lack the ability to fabricate the designed IC, we did not follow this avenue any further.

Later in the design process, we decided to use industrial level CAD tools such as the Edge toolset from Cadence, Lsim from Mentor Graphics, and Hspice from Metasoftware. The main driving force behind redoing the schematic entry and simulations was the need to ensure that the university level tools previously used had not hidden any circuit problems. The complete set of schematics is now in two different database formats: Cadence and Oct. The simulations showed that the circuit works as expected.

6.2 A Simulation Example

The simulation step was automated so as to allow for multiple runs with as little user intervention as possible. The Makefile shown in appendix D was used to run all RNL simulations, a similar Makefile was also written for running MUSA and Lsim simulations. A simulation step output is shown in appendix E. In it we can see how the Makefile is used to translate the netlist into a format readable by RNL, and how the vector generator program is compiled. The vector generator is then used to start a pipeline of programs that will produce the simulation results. [27]

The following data was extracted from actual simulation results. Some results have been annotated to make them readable and some have been condensed to a general case that conveys the same information as hundreds of vectors. We will go over the general simulation format of a PE instruction, in particular, we begin with the instruction loading cycle.

The required I/O signals were split into signal groups, called vectors, to make them manageable. The following simulation vectors were defined for all runs:

- reset = Signal used to clear all internal latches and set them to a known state.
- load = Signal used to load the IR on a 1 t-> 0 edge transition.
- ctl = A control vector formed by the reset and load signals in that order.
- r = Bits that control ALU output routing. They only have routing values when the IR is being loaded, otherwise they have data values placed on the corresponding bits of the A bus.

---

[27] Note how the RNL simulator prints out the number of devices found in the input netlist

- i = Bits that control ALU instruction. They only have valid op-codes when the IR is being loaded, otherwise they have data values placed on the corresponding bits of the A bus.
- req = The request vector formed by the reqa, reqb, reqc, and reqd signals in that order.
- reqin = The input request vector formed by the reqa and reqb signals in that order
- reqout = The output request vector formed by the reqc and reqd signals in that order.
- ack = The acknowledge vector formed by the acka, ackb, ackc, and ackd signals in that order.
- ackin = The input acknowledge vector formed by the ackc and ackd signals in that order
- ackout = The output acknowledge vector formed by the acka and ackb signals in that order.
- a = Input bus A
- b = Input bus B
- c = Output bus C
- d = Output bus D

The simulation step is composed of a vector value assignment step and a node value evaluation step The evaluation step allows the simulator to propagate the input vector values across the entire circuit network. The first step in programming the PE is to bring all latches to a known state, so we toggle the reset signal from high to low and set all needed input lines to low for consistency purposes. These two simulation steps generate the following result:

- 1 ctl=0b10 r=0x0 i=0x0 req=0bXX ack=0bXX00 a=0x0 b=0x0 c=0x0 d=0x0
- 2 ctl=0b00 r=0x0 i=0x0 req=0b0000 ack=0b0000 a=0x0 b=0x0 c=0x0 d=0x0

Line 1 sets the ctl vector to the hex value 0x2, which means that the reset signal is high. Note how the req vector is set to 0xX because bits 0 and 1 are set to X. This is due to the bits being driven by the output of rsff latches, which are in an unknown state. The same is true for bits 2 and 3 in the ack vector. Line 2 shows how the X states are set low after the reset has a falling edge.

At this point the control unit in the core is seeing a 1 operand instruction in the IR. The instruction in the IR is 00000000 after the reset, which is the inc_a, route outputs straight opcode (refer to table 7 for a list of valid op-codes, and table 11 for a list of ALU output routings).

Everything seems to be working well because reset succeed and the inc_a is a 1 operand instruction. The next step is to load the IR with a valid instruction. Our sample simulation worked on the ADD instruction. The load instruction cycle begins when we set the load signal high, set the a bus to ADD (0x29), and set the b bus to LOAD_PE_IR (0x2). Then we toggle the load signal from high to low which latches the contents of the a bus into the IR:

- 3 ctl=0b01 r=0x0 i=0xU req=0b0011 ack=0b1100 a=0xU b=0x2 c=0x0 d=0x0 [28]
- 4 ctl=0b00 r=0x0 i=0xU req=0b0011 ack=0b1100 a=0xU b=0x2 c=0x0 d=0x0

We can now try different combinations on the inputs and examine the outputs for correctness. Since our computation takes place in an asynchronous manner, we first want to make sure that the synchronization is working fine. We start by placing different invalid synchronization vectors along with different operand values on the A and B busses. When the reqc and reqd signals are turned on, along with the acka and ackb: No synchronization is taking place therefore no activity is shown on the outputs, even if the inputs change:

- 5 ctl=0b00 r=0xV i=0xW req=0b0011 ack=0b1100 a=0xY b=0xZ c=0x0 d=0x0 [29]

The results show that even though we changed the input values, they were not reflected on the output because the I/O synchronization is not right. The current synchronization tells us that both the req and consumed signals are low, meaning that there is no data to be consumed and that the previous result generated by this core has not been consumed.

Then the ackd signal arrives but nothing changes, except that the reqd and ackb get reset by the arrival of the ackd signal:

- 6 ctl=0b00 r=0xV i=0xW req=0b0010 ack=0b1001 a=0xY b=0xZ c=0x0 d=0x0

Later the acke signal arrives but nothing changes, except that the reqc and acka get reset by the arrival of the acke signal:

- 7 ctl=0b00 r=0xV i=0xW req=0b0000 ack=0b0000 a=0xY b=0xZ c=0x0 d=0x0

There is no change on the output busses because the arrival of the consumed signal only means that the previous result was consumed. Since the req signal is not present, the core assumes that any data on its input busses is not valid. The arrival of the second ack completes the output synchronization and the output

---

28 Where U is a hex digit in the instruction set. For this sample simulation U = 29

29 Where V is a hex digit in {0...3}, W is a hex digit in {00...3F}, Y is a hex digit in {00...FF}, and Z is a hex digit in {00...FF}

synchronization latches are returned to zero. The syncout module generates a consumed signal for the ALU, based on arrival of both ack signals. The output synchronization is also valid if the acke arrives before the ackd or they both arrive simultaneously. The synchronization protocol is not sensitive to signal ordering.

Given that the output synchronization is already complete, then the following two general vectors should bring us to the required I/O synchronization:

- 8 ctl=0b00 r=0xV i=0xW req=0b1000 ack=0b0000 a=0xY b=0xZ c=0x0 d=0x0
- 9 ctl=0b00 r=0xV i=0xW req=0b1100 ack=0b0000 a=0xX b=0xX c=0xX d=0xX

Input synchronization is also insensitive to signal ordering. This step brings us to the required synchronization; however, the path shown above for getting to the required synchronization is only one of many possible paths. A path in this case is defined as the order in which the required synchronization signals arrive, and since our protocol is completely insensitive to signal ordering then all possible paths are equivalent.

The last step is to show the PE executing an instruction with all of the required synchronization present:

- 10 ctl=0b00 r=0xV i=0xW req=0b1111 ack=0b1100 a=0xY b=0xZ c=0xR d=0xS
- 11 ctl=0b00 r=0xV i=0xW req=0b0011 ack=0b1100 a=0xY b=0xZ c=0xR d=0xS [30]

Line 10 shows how the core starts an output synchronization cycle, by setting the reqout vector high. The core also acknowledges the current inputs being used to its input partners, thus concluding the input synchronization cycle by setting the ackout vector high. Line 11 shows the input partners bringing the reqin vector low, which allows the core to become ready for more inputs. The core can generate results, have them latched into its output busses, and go ahead to work on the next set of inputs even as it is waiting to have the previous results consumed. Refer to appendix F for a more complete, but partial set of simulation results.

---

30 Where R is A + B, and S is a Partial Status

Chapter 7 Conclusion

We believe that the contents of this thesis can be successfully used to build a prototype array architecture. The main thrust of the research should now focus on software-related issues such as developing a compiler for this architecture and also fabricating a prototype PAAP ICs.

As this project goes forth, and a detailed model for the PAAP machine architecture is explored, the restrictions placed on the machine architect by the PAAP element may be well-understood. It is at this point that further research might be done to provide better hardware support for the proposed architecture. Any enhancement to the original PAAP element design is beyond the scope of this thesis, however, we will discuss possible areas for improvements:

- The available on-board register file may be better used if it were consolidated into a FIFO-type cache. This way, multiple data bytes could be received from the PP and distributed to their destination only when it is safe.

  In the current design, the HOST has to communicate with the PAAP addresses one address range at the time. While it is true that a block of addresses and data could be sent to the PAAP by the HOST, there is currently no cache system to aid in the possible transfer bottleneck.

- The routing element should be provided with an extra capability so that it can broadcast the contents of one of its busses to all other busses connected to it.

  At the moment this capability is not present and it might require a complete RE redesign in order to provide it. There should also be a way to determine when the RE has a bad instruction setting. The present design assumes that a valid instruction is stored in the RE's IR, but will fail with no warning if the value stored in the IR is not valid.

- The PE's output busses should have a tri-state capability.

  This is useful while trying to optimize the usage of routing tracks. If the status is of no interest to the next computation then it should be tri-stated.

- The instruction set could implement a controlled shift instruction that would take one of its inputs as the parameter that tells it how many bits to shift its other parameter.

  In the current version, the PAAP provides both shift left/right instructions, but they are single-bit shifts of the A operand. There might also be the need to provide a rotate instruction derived from a modified shift operation.

The following points are concerns that must be addressed by the machine architect in order to properly use the PAAP design:

- The reset and CS circuitry should be activated on a rising edge.

  This is a matter of preference on the architecture designer's part. The PE does not make use of the CS signal for it is meant to select a sub-array.

- As the design is proven to be useful, we may want to experiment with a 32-bit system.

  The current design implements an 8-bit system, but a 32-bit system might be more appropriate for large applications. Every circuit element included in the PAAP design can be extended from 8 to 32 bits because most circuit elements were designed with expansion in mind.

- Our row and column loading techniques are vulnerable to WSI faults.

  WSI techniques could be used to construct large PAAP arrays, but there is a large possibility that faulty PEs or REs will be found at the wafer sort stage. This would break our program loading technique because there would be a break in the continuous row or column buses. As a result, only partial program loading could be done. The solution may involve a rethinking on our loading methodology, but this work is beyond the scope of this thesis

The PAAP Array illustrates the use of computation and routing units to make a highly parallel flexible architecture. Many questions remain, such as the relationship of data storage and I/O control for the array. The work has also raised even more fundamental points:

- What is the optimal size of an array?
- What is the optimal size of a PE and RE?
- What is the optimal RE flexibility?

The present work explored the concept, and demostrated its programmability. The next step will be to explore the above questions, and to determine whether extensive sets of PAAP type arrays could be an effective use of silicon in a future computer.

List of References

[AbelsonH84] Belson H., and Susman G., "Structure and Interpretation of Computer Programs", Cambridge, MA: MIT Press, 1984.

[ArvindamS90] Arvindam S., Kumar V., and Nagashwara V.R., "Efficient Parallel Algorithms for Search Problems: Applications in VLSI CAD", 3rd Symposium on the Frontiers of Massively Parallel Computation, IEEE, pp. 166-169, 1990.

[BalakrishnanM88] Balakrishnan M., Jain R., and Raghavendra C.S., "On Array Storage for Conflict-free Access for Parallel Processors", Proc. 17th International Conference on Parallel Processing, pp. 103-107, 1988.

[Balance84] The Balance 8000 System Technical Summary, Sequent Computer Systems Inc., 1984.

[BasuA88] Basu A., "A Classification of Parallel Processing Systems", Proc. of the IEEE Conference on Computer Design, pp. 222-225, 1984.

[BitarP86] Bitar P. and Despain A.M., "Multiprocessor Cache Synchronization-Issues, Innovations, Evolution", IEEE Conference Proc. on Computer Architecture, pp. 424-433, 1986.

[BrozozowskiJ89] Brozozowski J. and Ebergen J.C., "Recent Developments in the Design of Asynchronous Circuits", Technical Report CS-89-18, Department of Computer Science, University of Waterloo, 1989.

[DeCegamaA(a)89] DeCegama A.L., "The technology of Parallel Processing", New Jersey: Prentice Hall, pp. 308-316, 1989.

[DeCegamaA(b)89] DeCegama A.L., pp. 463-466

[DelzompoA89] Delzompo A., "UNIX Networking: NFS and RPC", Hayden Books, Indianapolis, Indiana, pp. 93-120, 1989.

[DenningP86] Denning P.J., "Parallel Computer and Its Evolution", Communications of the ACM, Vol. 29, No. 12, pp. 1163-1167, 1986.

[DuncanR90] Duncan R., "A Survey of Parallel Computer Architectures", Computer, pp. 5-16, February 1990.

[FahlmanS87] Fahlman S.E. and Hinton G.E., "Connectionist Architectures for Artificial Intelligence", IEEE Computer, pp. 100-109, January 1987.

[FisherJ84] Fisher J.A., "The VLIW Machine: A Multiprocessor for Compiling Scientific Code", IEEE Computer, pp. 45-53, 1984.

[FlynnM56] Flynn M.J., "Very High Speed Computing Systems", Proc. IEEE Vol. 54, pp. 1901-1909, 1956.

[FullerS78] Fuller S.H. and Harbison S.P., "The C.mmp Multiprocessor", Technical Report CMU-CS-78-146, CMU, Computer Scince Department, 1978

List of References

[GP88] Overview of the Butterfly GP1000, BBN Advanced Computers Inc., November 1988.

[GannonD86] Gannon D., "Restructuring the SIMPLE for the CHiP architecture", Parallel Computing Vol 3, North-Holland: Elsevier Science Publishers, pp. 305-326, 1988.

[GopalakrishnanG90] Gopalakrishnan G. and Jain P., "Some Recent Asynchronous System Design Methodologies", Technical Report UU-CS-TR-90-016, Department of Computer Science, University of Utah, 1990.

[GottliebA82] Gottlieb A. and Schwartz J.T., "Networks and Algorithms for Very-Large-Scale Parallel Computation", IEEE, Computer, Vol. 15, No. 1, pp. 27-34, Jan 1982.

[HarrisonD86] Harrison D.S., et al., "Data Management and Graphics Editing in the Berkeley Design Environment", Proc. of the International Conference on CAD, IEEE, pp. 20-34, 1986.

[HennessyJ91] Hennessy J. L. and Joupi N.P., "Computer Technology and Architecture: An Evolving Interaction", IEEE Computer Vol. 24, No. 9, pp. 18-28, 1991

[HordM(a)90] Hord R.M., "Parallel Supercomputing in SIMD Architectures", New Jersey: CRC Press, pp. 5-9, 1990.

[HordR(b)90] Hord R.M., pp. 17-84

[HordR(c)90] Hord R.M., pp. 205-300

[KatevenisM84] Katevenis M.G.H., "Reduced Instruction Set Computer Architectures for VLSI", Boston, MA: MIT Press, 1984.

[KruatrachueK88] Kruatrachue K. and T. Lewis, "Grain Size Determination for Parallel Processing", IEEE Computer, pp. 23-32, January 1988.

[KorenI88] Koren I., Mendelson B., and Peled I., "A Data-Driven VLSI Array for Arbitrary Algorithms", IEEE Computer, pp. 30-43, October 1988.

[KuckD86] Kuck D.J. et al., "Parallel Supercomputing Today and the Cedar Approach", Science, Vol. 2, pp. 967-974, 1986.

[KumarV87] Kumar V. and Negashwara V.R., "Parallel Depth-first Search Part II: Analysis", International Journal in Parallel Programming, 16(6), pp. 501-519, 1987.

[KumarV90] Kumar V. and Negashwara V.R., "Scalable Parallel Formulations of Depth-first Search", New York, NY: Springer-Verlag, 1990.

[KungH78] Kung H.T. and Leiserson C.E., "Systolic Arrays for VLSI", Sparse Matrix Symposium, SIAM, pp. 256-282, 1978.

[KungH82] Kung H.T., "Why Systolic Architectures?", IEEE, Computer, Vol. 15, No. 1, pp. 37-46, Jan 1982.

List of References

[KungS82] Kung S. Y. and Gol-Ezer R. J., "Synchronous vs. Asynchronous Computation in VLSI Array Processors", IEEE, SPIE Proc. pp. 232-243, 1982.

[KungS85] Kung S.Y. and Kumar V.K.P., "Wavefront Array Processor and Beyond", IEEE Conference on Computer Design, pp. 176-179, 1985.

[KungS88] Kung S.Y., "VLSI Array Processors", Englewood Cliffs, New Jersey: Prentice Hall, 1988.

[MapplesC85] Mapples C., "Pyramids, Crossbars and Thousands of Processors", Proc. IEEE International Conference of Parallel Processing, pp. 681-688, 1985.

[MelvinS87] Melvin S.W. and Patt Y.N., "A Clarification of the Dynamic/Static Interface", Proc. 20th Int'l Conf. System Sciences, IEEE, pp. 218-226, 1987.

[MohanJ83] Mohan J., et al., "Granularity of Parallel Computation", CMU, Computer Science Department Technical Report, April 1983.

[NagashwaraV87] Nagashwara V.R. and Kumar V., "Parallel Depth-first Search Part I: Implementation", International Journal in Parallel Programming, 16(6), pp. 479-499, 1987.

[PolleyH91] Polley H., Unpublished Doctorate Seminar Presentation, Concordia University, Department of Computer Science, 1991.

[PeaseD91] Pease D., et al., "PAWS: A Performance Evaluation Tool For Parallel Computer Systems", IEEE Computer Vol. 24. No. 1, pp. 18-28, 1991.

[RettbergR90] Rettberg R.D. et al., "The Monarch Parallel Processor Hardware Design", IEEE Computer, Vol. 23, No. 4, 1990.

[RNL83] UW/NW VLSI Consortium, "RNL 4.2 User's Guide", University of Washington, Seattle, WA, pp. 1-33, 1983.

[SakaiS90] Sakai S., Kodama Y. and Yamagichi Y., "Prototype Implementation of a Highly Parallel Dataflow Machine: EM-4", Proc. of the 5th Int'l Parallel Processing Symposium, IEEE, pp. 278-286, 1990.

[SantoB88] Santo B. and Wollanrd K., "The World of Silicon", IEEE Spectrum, Vol. 25, No 9, pp. 30-39, 1988.

[SavariaY(a)86] Savaria Y. et al., "Soft-Error Filtering: A Solution to the Reliability Problem of Future VLSI Digital Circuits", IEEE Proc. Vol. 74, No. 5, pp. 669-683, May 1986.

[SavariaY(b)86] Savaria Y. et al., "A Theory for the Design of Soft-Error-Tolerant VLSI Circuits", IEEE Journal on Selected Topics in Communication, Vol. SAC-4, No. 1, pp. 15-33, Jan 1986.

[SchendelU84] Schendel U., "Introduction to Numerical Methods for Parallel Computers", Ellis Horwood Ltd., Chichester, England, 1984.

List of References

[SnyderL81] Snyder L., "Programming Processor Interconnection Structures", Technical Report CSD-TR381, Purdue University, 1981.

[SnyderL82] Snyder L, "Introduction to the Configurable Highly Parallel Computer", IEEE, Computer, Vol. 15, No. 1, pp. 47-56, Jan 1982.

[SriniV86] Srini V.P., "An Architectural Comparison of Data Flow Systems", Computer, pp. 68-87, March 1986.

[SUayaR90] Suaya R. and Birtwistle G., VLSI and Parallel Computation, San Mateo, CA: Kaufmann Publishers, 1990.

[SutherlandJ89] Sutherland J.E., "Micropipelines", Communications of the ACM, Vol 32, No. 6, pp. 720-738, 1989.

[TC89] TC2000 Technical Prodcut Summary, BBN Advanced Computers Inc., November 1989.

[TTLBook85] Texas Instrumets, The TTL Data Book, Dallas, Texas, pp. 3-709 - 3-720, 1895.

[TreleavenP88] Treleaven P.C., "Parallel Architecture Overview", Parallel Computing Vol 8, North-Holland: Elsevier Science Publishers, p. 59, 1988.

[VoltmerF85] Voltmer F.W. and Jones N.W., "Factors Contributing to Increased VLSI Density", VLSI Handbook, Chapter 1, Florida: Academic Press, 1985.

[Webers88] Weber S., "Will Quantom-Effect Technology Present a Quantom Jump in ICs?", Electronics, Vol. 61, No. 16, pp. 143-146, October 1986.

[YungH88] Yung H.Y. and Singh A.D., "A Highly Efficient Design for Reconfiguring the Processor Array in VLSI", Proc. of the International Conference on Parallel Processing, pp. 375-38, 1988.

APPENDIX A

Programming The PAAP

- Initialize the hardware
  - Toggle the Reset signal from 0 -> 1 -> 0

    This will set all internal registers to a known state. This is only supposed to be done upon power-up programming. If the PAAP is being reprogrammed it is not necessary to do this step.

- Write control word
  - Set the hex value 0x00 on the address bus

    We are addressing the control register R0.

  - Set the hex value 0x41 on the I/O bus

    This will effectively set C0 = 1 (we are loading), C6 = 1 (Remember address values), and C7 = 0 (we are writing).

  - Toggle the CS signal from 0 -> 1 -> 0

    This will load the value on the I/O bus into R0 (The control register).

- Initialize all Input registers to 0
  - Set the hex value 0x1F on the address bus.

    This will signal that a block transfer is being requested. Since the address has been latched, no more address setting is required during this block transfer.

  - Set the hex value 0x00 on data bus 1 - 10
  - Toggle the CS signal from 0 -> 1 -> 0

    This will load the value on data busses 1 to 10 into registers 2 to 11 respectively. In effect they will all be initialized to 0.

- Load instructions for PEs and REs in column 5
  - Set the instructions meant for the PEs in column 5 into data busses 1, 3, and 5.

- Set the instructions meant for the REs in column 5 into data busses 2 and 4.
- Toggle the CS signal from 0 -> 1 -> 0

  This will load the value on data busses 1, 3, and 5 into registers 2, 4 and 6 respectively. It will also load data busses 2 and 4 into registers 3 and 5 respectively.
- Set the hex value 0x06 on data bus 10
- Toggle the CS signal from 0 -> 1 -> 0

  This will set the instruction registers in the PEs and REs in column 5 to begin sampling their input bus A for values to be stored.
- Wait for 4-column propagation delays

  This will allow for the values stored in registers 2, 3, 4, 5, and 6 to be propagated all the way to the input busses of all elements in column 5.
- Set the hex value 0x04 on data bus 10
- Toggle the CS signal from 0 -> 1 -> 0

  This will load the value on data bus 10 into register 11 and load all instruction registers in all PEs in column 5.
- Set the hex value 0x00 on data bus 10
- Toggle the CS signal from 0 -> 1 -> 0

  This will load the value on data bus 10 into register 11 and load all instruction registers in all REs in column 5.

- Load instructions for REs in column 4
  - Set the instructions meant for the REs in column 4 into data busses 1, 2, 3, 4, and 5.
  - Toggle the CS signal from 0 -> 1 -> 0

    This will load the value on data busses 1, 2, 3, 4, and 5 into registers 2, 3, 4, 5 and 6 respectively.
  - Set the hex value 0x04 on data bus 9

- Toggle the CS signal from 0 -> 1 -> 0

  This will set the instruction registers in REs in column 4 to begin sampling their input bus A for values to be stored.

- Wait for 3-column propagation delays

  This will allow for the values stored in registers 2, 3, 4, 5, and 6 to be propagated all the way to the input busses of all REs in column 4.

- Set the hex value 0x00 on data bus 9

- Toggle the CS signal from 0 -> 1 -> 0

  This will load the value on data bus 9 into register 10 and load all instruction registers in all REs in column 4.

- Load instructions for PEs and REs in column 3

  - Set the instructions meant for the PEs in column 3 into data busses 1, 3, and 5.

  - Set the instructions meant for the REs in column 3 into data busses 2 and 4.

  - Toggle the CS signal from 0 -> 1 -> 0

    This will load the value on data busses 1, 3, and 5 into registers 2, 4 and 6 respectively. It will also load data busses 2 and 4 into registers 3 and 5 respectively.

  - Set the hex value 0x06 on data bus 8

  - Toggle the CS signal from 0 -> 1 -> 0

    This will set the instruction registers in PEs and REs in column 3 to begin sampling their input bus A for values to be stored.

  - Wait for 2-column propagation delays

    This will allow for the values stored in registers 2, 3, 4, 5, and 6 to be propagated all the way to the input busses of all elements in column 3.

  - Set the hex value 0x04 on data bus 8

  - Toggle the CS signal from 0 -> 1 -> 0

This will load the value on data bus 8 into register 9 and load all instruction registers in all PEs in column 3.

- Set the hex value 0x00 on data bus 8
- Toggle the CS signal from 0 -> 1 -> 0

  This will load the value on data bus 8 into register 9 and load all instruction registers in all REs in column 3.

- Load instructions for REs in column 2
  - Set the instructions meant for the REs in column 2 into data bus 1, 2, 3, 4, and 5.
  - Toggle the CS signal from 0 -> 1 -> 0

    This will load the value on data busses 1, 2, 3, 4, and 5 into registers 2, 3, 4, 5 and 6 respectively.

  - Set the hex value 0x04 on data bus 7
  - Toggle the CS signal from 0 -> 1 -> 0

    This will set the instruction registers in REs in column 2 to begin sampling their input bus A for values to be stored.

  - Wait for 1-column propagation delay

    This will allow for the values stored in registers 2, 3, 4, 5, and 6 to be propagated all the way to the input busses of all REs in column 2.

  - Set the hex value 0x00 on data bus 7
  - Toggle the CS signal from 0 -> 1 -> 0

    This will load the value on data bus 7 into register 8 and load all instruction registers in all REs in column 2.

- Load instructions for PEs and REs in column 1
  - Set the instructions meant for the PEs in column 1 into data busses 1, 3, and 5.
  - Set the instructions meant for the REs in column 1 into data busses 2 and 4.

- Toggle the CS signal from 0 -> 1 -> 0

  This will load the value on data busses 1, 3, and 5 into registers 2, 4 and 6 respectively. It will also load data busses 2 and 4 into registers 3 and 5 respectively.

- Set the hex value 0x06 on data bus 6

- Toggle the CS signal from 0 -> 1 -> 0

  This will set the instruction registers in PEs and REs in column 1 to begin sampling their input bus A for values to be stored.

- Wait for 0-column propagation delay

  This will allow for the values stored in registers 2, 3, 4, 5, and 6 to be propagated all the way to the input busses of all elements in column 1.

- Set the hex value 0x04 on data bus 6

- Toggle the CS signal from 0 -> 1 -> 0

  This will load the value on data bus 6 into register 7 and load all instruction registers in all PEs in column 1.

- Set the hex value 0x00 on data bus 6

- Toggle the CS signal from 0 -> 1 -> 0

  This will load the value on data bus 6 into register 7 and load all instruction registers in all REs in column 1.

APPENDIX B

Automatic Vector Generation Setup Files

main.h

```
/*****************************************************************************
```

The Programable Asynchronous Array Processor (PAAP) project is my research thesis. The work herein contained is submitted in partial fulfillment of the Master of Computer Science degree by Marco Zelada to the Computer Science Department, Engineering and Computer Science Faculty, Concordia University, Montreal, Canada.

```
*****************************************************************************
```

This work was done under the supervision of Dr. Terill Fancott, Associate Dean, Engineering and Computer Science Faculty. The work was conducted while at three different location, Concordia University, Intel Corporation, and Microchip Technology Inc. The companies involved were notified of the work in progress for this thesis when Marco A. Zelada joined. All Copyrights to this work are reserved by the author and Concordia University.

```
*****************************************************************************/
```

/* These conditional statements are here to be able to use the same code to generate simulator inputs for both rnl and musa. */

```
#ifdef MUSA
#include <musa.h>          /* Get the musa macro definitions */
#else
#ifdef RNL
#include <rnl.h>           /* Get the rnl macro definitions */
#else
#include <lsim.h>          /* Get the lsim macro definitions */
#endif
#endif

#include <instructions.h>  /* Get the instruction definitions */
#include <route.h>         /* Get the routing definitions */
#include <stdio.h>         /* Get the standard I/O stuff */

#define LIM0 0
#define LIM1 1
#define LIM2 3
#define LIM3 7
#define LIM4 15
#define LIM5 31
#define LIM6 63
#define LIM7 127
```

```
#define LIM8 255
#define LIM9 511
#define LIM10 1023
#define LIM11 2047
#define LIM12 4095
#define LIM13 8191
#define LIM14 16383
#define LIM15 32767
#define LIM16 65535
int value;                  /* Value to convert in the hex macro */
char *type;                 /* Type of vector, bin, oct, hex, dec */
char *types;                /* Type of vector, bin, oct, hex, dec */
char *signal;               /* Signal name in the hex macro */
char *signals;              /* Signal name in the hex macro */
char *signals2;             /* Signal name in the hex macro */
char *signals3;             /* Signal name in the hex macro */
char *signals4;             /* Signal name in the hex macro */
char *signals5;             /* Signal name in the hex macro */
char *signals6;             /* Signal name in the hex macro */
char *signals7;             /* Signal name in the hex macro */
char *signals8;             /* Signal name in the hex macro */
```

musa.h

```
/* Create the general setup */
#define setup( signal ){ }
/* Quit because we are done */
#define quit(){printf("exit\n");}
/* Take the signals out of the input list */
#define unset( signal ){\
printf(" se %s H000\n", signal);\
printf(" ev\n");\
printf(" se %s Hxxx\n");\
printf(" ev\n");}
/* Evaluate the result after having set the signal values */
#define evaluate( ){printf(" ev\n");}
/* Set up the vector naming convention */
#define vector( type, signal, signals ){ \
printf("makevector %s %s\n", signal, signals);}
/* Set the input vector */
#define set( value, signal ){\
```

```
if ( value <= 15 )\
        printf(" se %s H0%x\n", signal, value);\
else\
        printf(" se %s H%x\n", signal, value);}
/* Set up the output report format for signals */
#define format( signals ){\
printf("show \"\\n\"\n"); \
printf("show %s \n", signals);}
#define format2( types, signal ){\
printf("show \"\\n\"\n"); \
printf("show %s %s \n", types, signal);}
#define format3( types, signal, signals ){\
printf("show \"\\n\"\n"); \
printf("show %s %s %s \n", types, signal, signals);}
#define format4( types, signal, signals, signals2 ){\
printf("show \"\\n\"\n"); \
printf("show %s %s %s %s \n", types, signal, signals, signals2 );}
#define format5( types, signal, signals, signals2, signals3 ){\
printf("show \"\\n\"\n"); \
printf("show %s %s %s %s %s \n", types, signal, signals, signals2, signals3 );}
#define format6( types, signal, signals, signals2, signals3, signals4 ){\
printf("show \"\\n\"\n"); \
printf("show %s %s %s %s %s %s \n", types, signal, signals, signals2, signals3, signals4 );}
#define format7( types, signal, signals, signals2, signals3, signals4, signals5, ){\
printf("show \"\\n\"\n"); \
printf("show %s %s %s %s %s %s %s \n", types, signal, signals, signals2, signals3, signals4, signals5 );}
#define format8( types, signal, signals, signals2, signals3, signals4, signals5, signals6 ){\
printf("show \"\\n\"\n"); \
printf("show %s %s %s %s %s %s %s %s \n", types, signal, signals, signals2, signals3, signals4, signals5
signals6 );}
#define format11( type, types, signal; signals, signals2, signals3, signals4, signals5, signals6, signals7,
signals8 ){\
printf("show \"\\n\"\n"); \
printf("show %s %s %s %s %s %s %s %s %s %s %s \n", type, types, signal, signals, signals2, signals3,
signals4, signals5 signals6, signals7, signals8 );}
/* Set up the output report fortmat for signals and vectors */
#define formats( signal, signals ){\
printf("show \"\\n\"\n"); \
printf("show %s %s \n", signal, signals);}
```

rnl.h

```
/* The incr value described here is the simulation step time interval in 0.1ns units. A value of 150 is a clock
period of 15.0ns
        The clock frquency is thus f = 1/15.0 E-09 = 66.7 MHz */
#define setup( signal ){\
printf(";\n; RNL initialization files\n;\n", signal );\
printf("(load \"uwstd.l\")\n");\
printf("(load \"uwsim.l\")\n");\
printf("(read-network \"%s.rnl\")\n", signal );\
printf("(setq incr 150)\n");}
/* Set the input vector */
 #define set( value, signal ){\
printf("( invec \'( %s %d ))\n", signal, value );}
/* Take the signals out of the input list */
#define unset( signal ){printf("(x \'( %s ))\n", signal );}
/* Quit because we are done */
#define quit(){printf("exit\n");}
/* Evaluate t. e result after having set the signal values */
#define evaluate(){printf("(s \'())\n");}
#define evaluatep(){printf("(step incr)\n");}
/* Set up the vector naming convention */
#define vector( type, signal, signals ){ \
printf("(defvec \'(%s %s %s ))\n", type, signal, signals);\
printf("(chflag \'(%s))\n", signals); }
/* Set up the output report fortmat for signals */
#define format( signals ){ \
printf("(def-report \'(\" TEST \" newline \"MARCO\" %s ) )\n", signals );}
#define format2( types, signal ){ \
printf("(def-report \'(\" TEST \" newline \"MARCO\" (vec %s) (vec %s) ) )\n", types, signal );}
#define format3( types, signal, signals ){ \
printf("(def-report \'(\" TEST \" newline \"MARCO\" (vec %s) (vec %s) (vec %s) ) )\n", types, signal, signals
);}
#define format4( types, signal, signals, signals2 ){ \
printf("(def-report \'(\" TEST \" newline \"MARCO\" (vec %s) (vec %s) (vec %s) (vec %s) ) )\n", types,
signal, signals, signals2 );}
#define format5( types, signal, signals, signals2, signals3 ){ \
printf("(def-report \'(\" TEST \" newline \"MARCO\" (vec %s) (vec %s) (vec %s) (vec %s) (vec %s) ) )\n",
types, signal, signals, signals2, signals3 );}
#define format5s( types, signal, signals, signals2, signals3 ){\
printf("(def-report \'(\" TEST \" newline \"MARCO\" (vec %s) (vec %s) (vec %s) (vec %s) %s ) )\n", types,
signal, signals, signals2, signals3 );}
#define format6( types, signal, signals, signals2, signals3, signals4 ){\
```

```
printf("(def-report \'(\" TEST \" newline \"MARCO\" (vec %s) (vec %s) (vec %s) (vec %s) (vec %s) ) )\n", types, signal, signals, signals2, signals3, signals4 );}
#define format7( types, signal, signals, signals2, signals3, signals4, signals5 ){ \
printf("(def-report \'(\" TEST \" newline \"MARCO\" (vec %s) (vec %s) (vec %s) (vec %s) (vec %s) (vec %s) ) )\n", types, signal, signals, signals2, signals3, signals4, signals5 );}
#define format7s( type. , signal, signals, signals2, signals3, signals4, signals5 ){ \
printf("(def-report \'(\" TEST \" newline \"MARCO\" %s (vec %s) (vec %s) (vec %s) (vec %s) (vec %s) ) )\n", types, signal, signals, signals2, signals3, signals4, signals5 );}
#define format8( types, signal, signals, signals2, signals3, signals4, signals5, signals6 ){\
printf("(def-report \'(\" TEST \" newline \"MARCO\" (vec %s) (vec %s) (vec %s) (vec %s) (vec %s) (vec %s) (vec %s) ) )\n", types, signal, signals, signals2, signals3, signals4, signals5, signals6 );}
#define format9( types, signal, signals, signals2, signals3, signals4, signals5, signals6, signals7 ){\
printf("(def-report \'(\" TEST \" newline \"MARCO\" (vec %s) (vec %s) (vec %s) (vec %s) (vec %s) (vec %s) (vec %s) (vec %s) ) )\n", types, signal, signals, signals2, signals3, signals4, signals5, signals6, signals7 );}
#define format8s( types, signal, signals, signals2, signals3, signals4, signals5, signals6){\
printf("(def-report \'(\" TEST \" newline \"MARCO\" %s (vec %s) (vec %s) (vec %s) (vec %s) (vec %s) (vec %s) ) )\n", types, signal, signals, signals2, signals3, signals4, signals5, signals6);}
#define format11( type, types, signal, signals, signals2, signals3, signals4, signals5, signals6, signals7, signals8 ){\
printf("(def-report \'(\" TEST \" newline \"MARCO\" (vec %s) (vec %s) (vec %s) (vec %s) (vec %s) (vec %s) (vec %s) newline \"MARCO\" (vec %s) (vec %s) (vec %s) (vec %s) ) )\n", type, types, signal, signals, signals2, signals3, signals4, signals5, signals6, signals7, signals8 );}
/* Set up the output report fotmat for signals and vectors */
#define formats( signal, signals ){\
printf("(def-report \'(\" TEST \" newline \"MARCO\" (vec %s) %s ) )\n", signal, signals );}
#include <stdio.h>          /* Get the standard I/O stuff */
```

lsim.h

```
/* Create the general setup */
#define setup( signals ){                                    \
printf("\#\n\#\tGenerated stimulus for the %s circuit\n\#\n", signals );\
printf("\#\n\#\tClear all previous signal definitions\n\#\n");\
printf("Purge\n");                                                    \
printf("\#\n\#\tEnvironment Setup\n\#\n");\
printf("startup set Spike_Detection = OFF\n");\
printf("startup set Simulate_Spikes = OFF\n");\
printf("startup set Charge_Decay = NO\n");\
printf("startup set Decay_Time = 15 ns\n");\
printf("startup set Decay_Messages = OFF\n");\
printf("startup set Propagate_Unknowns = ON\n");\
printf("startup set Reduce_X_Pessimism = ON\n\n");\
```

```
printf("startup set RC_Timing = ON\n");\
printf("startup set Parallel_Trans = OFF\n");\
printf("startup set Timing_Checks = YES \n");\
printf("startup set Initial_Simulation = YES\n");\
printf("startup set Simulation_Interval = 500 ns\n");\
printf("startup set Printer_Name = LaserWriter\n");\
printf("startup timescale 20\n");\
printf("startup set Useri0 = 0\n");\
printf("startup set Useri1 = 10\n");\
printf("startup set Useri2 = 160\n");\
printf("startup set Userd0 = 10\n");\
printf("\#\n\#\tSet the simulation clock period to be 15 ns\n\#\n");\
printf("clockdef system 15\n");}

/* Set the input vector */
#define set( value, signal ){printf("setbus 0x%x %s\n", value, signal );}

/* Quit because we are done */
#define quit(){printf("exit\n");}

/* Evaluate the result after having set the signal values */
#define evaluate(){printf("simulate \n\n");}

/* Set up the vector naming convention */
#define vector( type, signal, signals ){ \
 printf("bus %s %s %s \n", signal, type, signals);}

/* Set a list of single bit probes */
#define probe( signals ) {\
 printf("probe %s\n", signals ); }

/* Delete the list of probes from the display */
#define delete( signals ){              \
 printf("hide %s\n", signals );}

/* Stop forcing a value on the list of probes */

#define unset( signals ){               \
 printf("release %s\n", signals );\
 printf("simulate \n" );}
```

instructions.h

/* This file has the definitions of the instructions the core can perform. The decimal numbers in the define statemets should be interpreted as follows:

i5	i4	i3	i2	i1	i0	
Cin	M	S3	S2	S1	S0	
0	0	X	X	X	X	The value is between 0 - 15, and we are in the arithmetic mode with carry.
0	1	X	X	X	X	The value is between 16 - 31, and we are in the logic mode.
1	0	X	X	X	X	The value is between 32 - 47, and we are in the arithmetic mode with no carry.
1	1	X	X	X	X	The value is between 48 - 63, and we are in the logic mode.

Once M = 1, Cin = X, because the mode is logic and there is no need for the carry. */

/* Zero operand instructions

	IR 543210	Hex	Dec
CLEAR	X10011	0x13	19
SET	X11100	0x1c	28
NEG	100011	0x23	35 */
#define CLEAR	19	/* The result is all 0's regardless of A or B */	
#define SET	28	/* The result is all 1's regardless of A or B */	
#define NEG	35	/* The result is -1 in 2's compliment */	

/* One operand instructions

	IR 543210	Hex	Dec
INC_A	000000	0x00	0
SHIFT_AC	001100	0x0c	12
NOT_A	X10000	0x10	16
NOT_B	X10101	0x15	21
PASS_B	X11010	0x1a	26
PASS_A	100000	0x20	32
SHIFT_A	101100	0x2c	44
SHIFTR_A	111111	0x3f	63
DEC_A	101111	0x2f	47 */
#define INC_A	0	/* The result is A + 1 */	
#define SHIFT_AC	12	/* The result is A shifted to the left by one position with 1 filling in the empty position */	
#define NOT_A	16	/* The result is A inverted */	

```
#define NOT_B       21          /* The result is B inverted */
#define PASS_B      26          /* The result is B */
#define PASS_A      32          /* The result is A */
#define SHIFT_A 44              /* The result is A shifted to the left by one position
                                   with 0 filling in the empty position */
#define SHIFTR_A    63          /* The result is A shifted to the right by one position
                                   with 0 filling in the empty position */
#define DEC_A       47          /* The result is A - 1 */

/* Two operand instructions
                    IR
                    543210      Hex         Dec
ORC                 000001      0x01        1
SUB                 000110      0x06        6
ADDC                001001      0x09        9
PASS_ABCOND         001111      0x0f        15
NOR                 X10001      0x11        17
NAND                X10100      0x14        20
XOR                 X10110      0x16        22
XNOR                X11001      0x19        25
AND                 X11011      0x1b        27
OR                  X11110      0x1e        30
PASS_AB             011111      0x1f        31
COMP_SUB            100110      0x26        38
ADD                 101001      0x29        41 */
#define ORC         1           /* The result is = A OR B + 1, Or with Carry */
#define SUB         6           /* The result is A - B */
#define ADDC        9           /* The result is A + B + 1, Add with Carry */
#define PASS_ABCOND     15      /* The result is A and status is B, if status bit 7 is set, 0
                                   else. This allows us to be able to test for any condition
                                   in the status bus by just shifting the bit we are insterested
                                   in into the 7th position. Multiple test can be done since
                                   both A and B are passed if the condition is true. */
#define NOR         17          /* The result is A NOR B */
#define NAND        20          /* The result is A NAND B */
#define XOR         22          /* The result is A XOR B */
#define XNOR        25          /* The result is A XNOR B */
#define AND         27          /* The result is A AND B */
#define OR          30          /* The result is A OR B */
#define PASS_AB     31          /* The result is A and status is B */
```

```
#define COMP_SUBC    38        /* The status is the result of a magnitud comparison is
                               placed in the status bus and. The result bus has A - B - 1,
                               Subtract with borrow */
#define ADD          41        /* The result is A + B */

/* These are repeated instructions. */
#define R1           3         /* Repeated CLEAR instruction */
#define R2           7         /* Repeated Misc 5, F = A AND NOT B */
#define R3           11        /* Repeated AND instruction */
#define R4           33        /* Repeated OR instruction */
#define R5           34        /* Repeated Misc 6, F = A OR NOT B */

/* These are by-product instructions with no meaning for our purposes */
#define M1           2         /* Misc 2, F = A OR NOT B + 1 */
#define M2           18        /* Misc 3, F = NOT A AND B */
#define M3           23        /* Misc 4, F = A AND NOT B */
#define M4           24        /* Misc 5, F = NOT A OR B */
#define M5           29        /* Misc 6, F = A OR NOT B */
#define M6           39        /* Misc 7, F = A AND NOT B - 1 */
#define M7           43        /* Misc 8, F = A AND B - 1 */
#define M8C          5         /* Misc 9C, F = (A OR B) + (A AND NOT B) + 1 */
#define M8           37        /* Misc 9, F = (A OR B) + (A AND NOT B) */
#define M9C          4         /* Misc 10C, F = A + (A AND NOT B) + 1 */
#define M9           36        /* Misc 10, F = A + (A AND NOT B) */
#define M10C         8         /* Misc 11C, F = A + (A AND B) + 1 */
#define M10          40        /* Misc 11, F = A + (A AND B) */
#define M11C         10        /* Misc 12C, F = (A OR NOT B) + (A AND B) + 1 */ #define
M11          42        /* Misc 12, F = (A OR NOT B) + (A AND B) */
#define M12C         13        /* Misc 13C, F = (A OR B) + A + 1 */
#define M12          45        /* Misc 13, F = (A OR B) + A */
#define M13C         14        /* Misc 14C, F = (A OR NOT B) + A + 1 */
#define M13          46        /* Misc 14, F = (A OR NOT B) + A */

/* General purpose constants */
#define CONDITION    0x01      /* The pass on condition is checked on b0 */
#define LOAD_PE_IR   0x02      /* Loading the PE IR is on b1 */
#define LOAD_RE_IR   0x04      /* Loading the Router IR is on b2 */
#define STRAIGHT     0x0       /* Output Routing */
#define BROADCAST_B  0x1       /* Output Routing */
#define BROADCAST_A  0x2       /* Output Routing */
#define CROSS        0x3       /* Output Routing */
```

route.h

/* The values here can be interpreted as follows:

D		C		B		A		Comment	
r7	r6	r5	r4	r3	r2	r1	r0	I/O config	Hex Value
1	1	1	0	1	1	1	0	a->c, b->d	0xEE
1	1	1	0	1	0	1	1	b->c, a->d	0xEB
1	1	1	1	0	1	0	1	a->b, c->d	0xF5
1	1	0	1	0	1	1	1	c->b, a->d	0xD7
1	0	1	0	0	1	0	1	a->b, d->c	0xA5
0	1	1	0	0	1	1	0	d->b, a->c	0x66
1	1	1	1	0	0	0	0	b->a, c->d	0xF0
1	1	0	0	1	1	0	0	c->a, b->d	0xCC
1	0	1	0	0	0	0	0	b->a, d->c	0xA0
0	0	1	0	1	0	0	0	d->a, b->c	0x28
0	1	0	0	0	1	0	0	c->a, d->b	0x44
0	0	0	1	0	1	0	0	d->a, c->b	0x14 */

```
#define  AtoC_AND_BtoD      0xEE
#define  BtoC_AND_AtoD      0xEB
#define  AtoB_AND_CtoD      0xF5
#define  CtoB_AND_AtoD      0xD7
#define  AtoB_AND_DtoC      0xA5
#define  DtoB_AND_AtoC      0x66
#define  BtoA_AND_CtoD      0xF0
#define  CtoA_AND_BtoD      0xCC
#define  BtoA_AND_DtoC      0xA0
#define  DtoA_AND_BtoC      0x28
#define  CtoA_AND_DtoB      0x44
#define  DtoA_AND_CtoB      0x14
```

Lsim pe.c

```
#include <main.h>

main()
{
int c0, c1, c2, c3, c4, c5, c6;

        setup( "pe" );
        probe("load reset reqa reqb ackc ackd reqc reqd acka ackb")

/* A and B are input busses, C and D are output buses */
```

```
        vector( "x", "a", "a[7] a[6] a[5] a[4] a[3] a[2] a[1] a[0]" );
        vector( "x", "b", "b[7] b[6] b[5] b[4] b[3] b[2] b[1] b[0]" );
        vector( "x", "c", "c[7] c[6] c[5] c[4] c[3] c[2] c[1] c[0]" );
        vector( "x", "d", "d[7] d[6] d[5] d[4] d[3] d[2] d[1] d[0]" );

/* We need to set the circuit into a known state, so we initialize
   all of the input signals.  All registers in the PE and Router will be
   cleared to 0 when the reset signal toggles = 1->0 */

        set( 1, "reset" );
        set( 0, "load");
        set( 0, "a" );
        set( 0, "b");
        set( 0, "reqa" );
        set( 0, "reqb" );
        set( 0, "ackc" );
        set( 0, "ackd" );
        evaluate();
        set( 0, "reset" );
        evaluate();

/* The value of c0 controls the instruction to be executed */

    for( c0=ADD; c0 <= ADD ;c0++ )
        switch ( c0 ){
                case CLEAR: /* Zero operand instructions */
                case SET:
                case NEG:
                case INC_A: /* One operand instructions */
                case SHIFT_AC:
                case SHIFT_A:
                case SHIFTR_A:
                case NOT_A:
                case NOT_B:
                case PASS_B:
                case PASS_A:
                case DEC_A:
                case ORC. /* Two operand instructions */
                case SUB:
                case ADDC:
                case NOR:
                case NAND:
                case XOR:
                case XNOR:
                case AND:
                case OR:
                case COMP_SUBC:
                case PASS_ABCOND:
                case PASS_AB:
                case ADD:
```

```
        for( c1=0; c1 <=0;c1++ )
         {
                set( (c0 | ( c1 << 6)), "a" );
                set( LOAD_PE_IR, "b" );/* Set b1=1, load IR */

/* Togle the load signal so that the IR gets loaded */

                for( c2=LIM1; c2 >= 0; c2-- )
                {
                        set( c2, "load" );
                        evaluate();
                }

                for( c3=LIM2; c3 <= LIM2; c3++ )
                 for( c4=LIM2; c4 <= LIM2; c4++ )
                 {
                        set( c3, "a" );
                        set( c4, "b" );

                        for( c5=0; c5 <= LIM1; c5++ )
                         for( c6=LIM1; c6 >= 0; c6-- )
                        {
                                set( c5, "reqa" );
                                set( c5, "reqb" );
                                set( c6, "ackc" );
                                set( c6, "ackd" );
                                evaluate();
                        }
        }/* end of for on c4 */
        }/* end of for on c1 */
                        break;
                default:
                        break;
        }/* end of switch */
        exit(0)
}
```

APPENDIX C

PAAP Detailed Circuit RNL Netlist

```
;       Non-inverting buffer
(macro buf (out in)
;       in      = Input
;       out     = Strenghten input
( local internal )
        ( cinvert internal in )
        ( cinvert out internal ))
;       Transmission gate that requires only one control signal
(macro ctgate (out in ctl)
;       in      = Input
;       ctl     = control signal
;       out     = Output = input when ctl = 1
( local bar )
        ( cinvert bar ctl )
        ( tgate out in ctl bar ))
;       The 2 Input AND gate
(macro cand (out a b)
;       a, b = Inputs
;       out = output
( local bar )
        ( cnand bar a b )
        ( cinvert out bar ))
;       The 3 Input AND gate
(macro cand3 (out a b c)
;       a, b, c = Inputs
;       out     = output
( local bar )
        ( cnand bar a b c )
        ( cinvert out bar ))
;       The 4 Input AND gate
(macro cand4 (out a b c d)
;       a, b, c, d = Inputs
;       out     = output
( local bar )
        ( cnand bar a b c d )
        ( cinvert out bar ))
;       The 5 Input AND gate
```

```
(macro cand5 (out a b c d e)
;       a, b, c, d, e = Inputs
;       out     = output
( local bar )
        ( cnand bar a b c d e )
        ( cinvert out bar ))
;       The 2 Input OR gate
(macro cor (out a b)
;       a, b = Inputs
;       out = output
( local bar )
        ( cnor bar a b )
        ( cinvert out bar ))
;       The 3 Input OR gate
(macro cor3 (out a b c)
;       a, b, c = Inputs
;       out     = output
( local bar )
        ( cnor bar a b c )
        ( cinvert out bar ))
;       The 4 Input OR gate
(macro cor4 (out a b c d)
;       a, b, c = Inputs
;       out     = output
( local bar )
        ( cnor bar a b c d )
        ( cinvert out bar ))
;       The XOR gate
(macro cxor (out a b)
;       a, b = Inputs
;       out = output
( local p0 p1 p2 )
        ( cnand p0 a b )
        ( cor p1 a b )
        ( cnand p2 p0 p1 )
        ( cinvert out p2 ))
;       1 Bit Switch box
(macro sbox ( c d a b c1 c2 )
;       c, d    = Outputs
;       a, b    = Inputs
```

```
;       c1, c2    = Control lines
( local c1n c2n p1 p2 p3 p4 )
        ( cinvert c1n c1 )
        ( cinvert c2n c2 )
        ( cnand p1 a c2n )
        ( cnand p2 b c2 )
        ( cnand c p1 p2 )
        ( cnand p3 b c1n )
        ( cnand p4 a c1 )
        ( cnand d p3 p4 ))
;       8 Bit Switch Box
(macro switch ( r1 r2 c0 c1 c2 c3 c4 c5 c6 c7 d0 d1 d2 d3 d4 d5 d6 d7 a0 a1 a2 a3 a4 a5 a6 a7 b0 b1 b2 b3 b4
b5 b6 b7 )
;       r1, r2    = Route control lines
;       a0-a7     = A Data ( usually the result from the ALU )
;       b0-b7     = B Data ( usually the status from the ALU )
;       c0-c7     =
;       d0-d7     = Outputs
        ( sbox c0 d0 a0 b0 r1 r2 )
        ( sbox c1 d1 a1 b1 r1 r2 )
        ( sbox c2 d2 a2 b2 r1 r2 )
        ( sbox c3 d3 a3 b3 r1 r2 )
        ( sbox c4 d4 a4 b4 r1 r2 )
        ( sbox c5 d5 a5 b5 r1 r2 )
        ( sbox c6 d6 a6 b6 r1 r2 )
        ( sbox c7 d7 a7 b7 r1 r2 ) )
;       Look-ahead Carry Generator
(macro lacg (cout cin p g )
;       p, g, cin = Inputs
;       cout              = Carry for the next stage
( local p0 p1 p2 )
        ( cinvert p0 cin )
        ( cand p1 p g )
        ( cand p2 p0 g )
        ( cnor cout p1 p2 ))
;       A 2-1 MUX
(macro mux2 (c r a b )
;       r         = Mux control
;       a, b      = Inputs
;       c         = Outputs
( local rn p1 p2 )
```

```
        ( cinvert rn r )
        ( cand p1 rn a )
        ( cand p2 r b )
        ( cor c p1 p2 ))
;       8 Bit 2-1 Mux
(macro mux ( c7 c6 c5 c4 c3 c2 c1 c0 r a7 a6 a5 a4 a3 a2 a1 a0 b7 b6 b5 b4 b3 b2 b1 b0 )
;       r       = Mux control
;       a0-a7   = A Data
;       b0-b7   = B Data
;       c0-c7   = Outputs
        ( mux2 c0 r a0 b0 )
        ( mux2 c1 r a1 b1 )
        ( mux2 c2 r a2 b2 )
        ( mux2 c3 r a3 b3 )
        ( mux2 c4 r a4 b4 )
        ( mux2 c5 r a5 b5 )
        ( mux2 c6 r a6 b6 )
        ( mux2 c7 r a7 b7 ))
;       The 74181 1 bit decode
(macro decode (dout low high a b s3 s2 s1 s0 )
;       dout = The output of the decode circuit
;       low = The output of the lower 3 ands and nor gates
;       high = The output of the upper 2 ands and nor
;
;       a, b = Data inputs a(i) b(i)
;       s0-s3 = Function selection bits
( local bn p0 p1 p2 p3 p4 )
        ( cinvert bn b )
        ( cand p0 a a )
        ( cand p1 s0 b )
        ( cand p2 s1 bn )
        ( cand3 p3 s2 bn a )
        ( cand3 p4 s3 b a )
        ( cnor low p0 p1 p2 )
        ( cnor high p3 p4 )
        ( cxor dout low high ))
;       The 74181 4 bit ALU
(macro alu41 ( f3 f2 f1 f0 c z eq p g cin m a3 a2 a1 a0 b3 b2 b1 b0 s3 s2 s1 s0 )
;       f0-f3 = Result
;       c = Carry out or Overflow
```

```
;       z = The result was Zero
;       eq = A = B signal
;       p, g = Carry generation for next stage
;       cin = Carry in
;       m = Mode selection Arithmetic/Logic
;       a0-a3 = A data
;       b0-b3 = B data
;       s0-s3 = ALU Function selection
( local p0 p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11 p12 p13 p14 p15 p16 p17 p18 p19 p20 mbar d0 d1 d2 d3 l0 l1
l2 l3 h0 h1 h2 h3 )
        ( cinvert mbar m )
;       Generate the F0 signal
        ( decode d0 l0 h0 a0 b0 s3 s2 s1 s0 )
        ( cnand p0 mbar cin )
        ( cxor f0 p0 d0 )
;       Generate the F1 signal
        ( decode d1 l1 h1 a1 b1 s3 s2 s1 s0 )
        ( cand p1 mbar l0 )
        ( cand3 p2 mbar cin h0 )
        ( cnor p3 p1 p2 )
        ( cxor f1 p3 d1 )
;       Generate the F2 signal
        ( decode d2 l2 h2 a2 b2 s3 s2 s1 s0 )
        ( cand p4 mbar l1 )
        ( cand3 p5 mbar l0 h1 )
        ( cand4 p6 mbar cin h0 h1 )
        ( cnor p7 p4 p5 p6 )
        ( cxor f2 p7 d2 )
;       Generate the F3 signal
        ( decode d3 l3 h3 a3 b3 s3 s2 s1 s0 )
        ( cand p8 mbar l2 )
        ( cand3 p9 mbar l1 h2 )
        ( cand4 p10 mbar l0 h1 h2 )
        ( cand5 p11 mbar cin h0 h1 h2 )
        ( cnor p12 p8 p9 p10 p11 )
        ( cxor f3 p12 d3 )
;       Generate P
        ( cnand p h0 h1 h2 h3 )
;       Generate the G
        ( cand4 p14 l0 h1 h2 h3 )
        ( cand3 p15 l1 h2 h3 )
```

```
        ( cand p16 l2 h3 )
        ( cand p17 l3 l3 )
        ( cnor g p14 p15 p16 p17 )
;       Generate the A = B
        ( cand4 eq f0 f1 f2 f3 )
;       Generate the Result = 0 flag
        ( cnor z f0 f1 f2 f3 ))
;       The 74181 4 bit ALU
(macro alu4 ( ack req f3 f2 f1 f0 c z eq p g cin m a3 a2 a1 a0 b3 b2 b1 b0 s3 s2 s1 s0 )
;       req = Request signal used for synchronization
;       ack = Acknowledge signal used for synchronization
;       f0-f3 = Result
;       c = Carry out or Overflow
;       z = The result was Zero
;       eq = A = B signal
;       p, g = Carry generation for next stage
;       cin = Carry in
;       m = Mode selection Arithmetic/Logic
;       a0-a3 = A data
;       b0-b3 = B data
;       s0-s3 = ALU Function selection
( local p25 p0 p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11 p12 p13 p14 p15 p16 p17 p18 p19 p20 p21 p22 p23 p24
mbar d0 d1 d2 d3 d4 l0 l1 l2 l3 l4 h0 h1 h2 h3 h4 l0b l1b l2b l3b h0b h1b h2b h3b cinb )
        ( cinvert mbar m )
;       Generate the F0 signal
        ( decode d0 l0 h0 a0 b0 s3 s2 s1 s0 )
        ( cnand p0 mbar cin )
        ( cxor f0 p0 d0 )
;       Generate the F1 signal
        ( decode d1 l1 h1 a1 b1 s3 s2 s1 s0 )
        ( cand p1 mbar l0 )
        ( cand3 p2 mbar cin h0 )
        ( cnor p3 p1 p2 )
        ( cxor f1 p3 d1 )
;       Generate the F2 signal
        ( decode d2 l2 h2 a2 b2 s3 s2 s1 s0 )
        ( cand p4 mbar l1 )
        ( cand3 p5 mbar l0 h1 )
        ( cand4 p6 mbar cin h0 h1 )
        ( cnor p7 p4 p5 p6 )
        ( cxor f2 p7 d2 )
```

```
;       Generate the F3 signal
        ( decode d3 l3 h3 a3 b3 s3 s2 s1 s0 )
        ( cand p8 mbar l2 )
        ( cand3 p9 mbar l1 h2 )
        ( cand4 p10 mbar l0 h1 h2 )
        ( cand5 p11 mbar cin h0 h1 h2 )
        ( cnor p12 p8 p9 p10 p11 )
        ( cxor f3 p12 d3 )
;       Generate the internal done signal
        ( decode d4 l4 h4 req req req req req req )
;       Buffer all crtitical path signals because we are exceeding some driver fanout
        ( buf h3b h3 )
        ( buf l3b l3 )
        ( buf h2b h2 )
        ( buf l2b l2 )
        ( buf h1b h1 )
        ( buf l1b l1 )
        ( buf h0b h0 )
        ( buf l0b l0 )
        ( buf cinb cin )
;       Create the longest path calculation
        ( cand p13 d4 l3b )
        ( cand3 p14 d4 l2b h3b )
        ( cand4 p15 d4 l1b h2b h3b )
        ( cand5 p16 d4 l0b h1b h2b h3b )
        ( cand5 p17 d4 h0b h1b h2b h3b )
        ( cand p18 p17 cinb )
        ( cnor p19 p13 p14 p15 p16 p18 )
;       The ack signal depends on the req signal as well as the longest delay path
        ( cxor p25 p19 d4 )
        ( cand ack p25 req )
;       Generate P
        ( cnand p h0 h1 h2 h3 )
:       Generate the Cout & G
        ( cnand p20 h0 h1 h2 h3 cin )
        ( cand4 p21 l0 h1 h2 h3 )
        ( cand3 p22 l1 h2 h3 )
        ( cand p23 l2 h3 )
        ( cand p24 l3 l3 )
        ( cnor g p21 p22 p23 p24 )
```

```
        ( cand c g p20 )
;       Generate the A = B
        ( cand4 eq f0 f1 f2 f3 )
;       Generate the Result = 0 flag
        ( cnor z f0 f1 f2 f3 ))
;       The 8 bit ALU
(macro alu ( ack req f7 f6 f5 f4 f3 f2 f1 f0 c z lt eq gt p g cin m a7 a6 a5 a4 a3 a2 a1 a0 b7 b6 b5 b4 b3 b2 b1
b0 s3 s2 s1 s0 )
;       req = Request signal used for synchronization
;       ack = Acknowledge signal used for synchronization
;       f0-f7 = Result
;       c = Carry out or Overflow
;       z = Zero result flag
;       lt = A < B signal *****These comparisons are only valid
;       eq = A = B signal*****
;       gt = A > B signal*****when the cin m s3 s2 s1 s0 = 38
;       p, g = Carry generation for next stage
;       cin = Carry in
;       m = Mode selection Arithmetic/Logic
;       a0-a7 = A data
;       b0-b7 = B data
;       s0-s3 = ALU Function selection
( local eq0 c0 p0 g0 eq1 cin1 p1 p2 p3 z0 z1 )
        (alu4l f3 f2 f1 f0 c0 z0 eq0 p0 g0 cin m a3 a2 a1 a0 b3 b2 b1 b0 s3 s2 s1 s0 )
        (lacg cin1 cin p0 g0 )
        (alu4 ack req f7 f6 f5 f4 c z1 eq1 p g cin1 m a7 a6 a5 a4 b7 b6 b5 b4 s3 s2 s1 s0 )
;       Generate the A = B
        ( cand eq eq0 eq1 )
;       Generate the A < B
        ( cinvert p1 c )
        ( cinvert p2 eq )
        ( cand3 lt cin p1 p2 )
;       Generate the A > B
        ( cinvert p3 lt )
        ( cand gt p2 p3 )
;       Generate the Zero result flag
        ( cand z z0 z1 ))
;       1 Bit bistable latch, it loads on a falling edge
(macro rsff (q s r load reset )
;       s, r = Inputs
;       load    = Loading and sync is not needed
```

```
;       reset     = Reset all latches to 0
;       q = Stored data
 ( local p0 p1 qnot )
        ( cnor p0 s load )
        ( cnor p1 r reset )
        ( cnand q p0 qnot )
        ( cnand qnot p1 q ))
;       1 Bit bistable latch, it loads on a falling edge, with reset
(macro dff (q clear d c)
;       d         = Data
;       c         = Enable
;       q         = Stored data
;       clear     = Reset the value stored to 0
( local p0 p1 p2 p3 p4 p5 )
        ( cinvert p0 d )
        ( cand p1 c p0 )
        ( cand p2 p3 p3 )
        ( cnor q p1 p2 clear )
        ( cand p4 d c )
        ( cand p5 q q )
        ( cnor p3 p4 p5 ))
;       8 Bit register
(macro register ( b7 b6 b5 b4 b3 b2 b1 b0 reset r0 a7 a6 a5 a4 a3 a2 a1 a0 )
;       reset     = Clear the register to all 0
;       r0        = Load Control
;       a0-a7     = Data to be stored
;       b0-b7     = Previously stored data
        ( dff b0 reset a0 r0 )
        ( dff b1 reset a1 r0 )
        ( dff b2 reset a2 r0 )
        ( dff b3 reset a3 r0 )
        ( dff b4 reset a4 r0 )
        ( dff b5 reset a5 r0 )
        ( dff b6 reset a6 r0 )
        ( dff b7 reset a7 r0 ))
;       8 Bit Shift Right register
(macro sregister ( b7 b6 b5 b4 b3 b2 b1 b0 reset ctl r0 a7 a6 a5 a4 a3 a2 a1 a0 )
;       reset     = Clear the register to all 0
;       ctl       = Shift Right
;       r0        = Load Control
```

```
;       a0-a7   = Data to be stored
;       b0-b7   = Previously stored data
(local ctlnot p0 p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11 p12 p13 p14 p15 a0r a1r a2r a3r a4r a5r a6r a7r)
        ( cinvert ctlnot ctl )
;       If shifting right fill in with 0s
        ( cnand p0 ctlnot a7 )
        ( cnand p1 ctl gnd )
        ( cnand a7r p0 p1 )
        ( cnand p2 ctlnot a6 )
        ( cnand p3 ctl a7 )
        ( cnand a6r p2 p3 )
        ( cnand p4 ctlnot a5 )
        ( cnand p5 ctl a6 )
        ( cnand a5r p4 p5 )
        ( cnand p6 ctlnot a4 )
        ( cnand p7 ctl a5 )
        ( cnand a4r p6 p7 )
        ( cnand p8 ctlnot a3 )
        ( cnand p9 ctl a4 )
        ( cnand a3r p8 p9 )
        ( cnand p10 ctlnot a2 )
        ( cnand p11 ctl a3 )
        ( cnand a2r p10 p11 )
        ( cnand p12 ctlnot a1 )
        ( cnand p13 ctl a2 )
        ( cnand a1r p12 p13 )
        ( cnand p14 ctlnot a0 )
        ( cnand p15 ctl a1 )
        ( cnand a0r p14 p15 )
        ( dff b7 reset a7r r0 )
        ( dff b6 reset a6r r0 )
        ( dff b5 reset a5r r0 )
        ( dff b4 reset a4r r0 )
        ( dff b3 reset a3r r0 )
        ( dff b2 reset a2r r0 )
        ( dff b1 reset a1r r0 )
        ( dff b0 reset a0r r0 ))
;       Control Unit, Generate the internal control signals
(macro cu ( i71 i61 i42 i32 i22 i12 i02 pass_ab shift_right 0op 1op 2op i7 i6 i5 i4 i3 i2 i1 i0 bn load )
;       load            = Load register signal
```

```
;       i7 - i0          = Instruction register contents.
;       i71 i61 i42 i32 i22 i12 i02
;                        = Adjusted instruction register value to force the
;                        PE to do a specific action, regarless of the
;                        instruction register contents.
;       shift_right= Shift A right 1 bit instruction
;       pass_ab = PASS A and B instruction control
;       bn               = Condition value to test by the pass on
;                        condition instruction.
;       0op, 1op, 2op= Signal the number of operands needed by
;                        the current instruction.
( local sr int_pass i0not i1not i2not i3not i4not i5not p0 p1 p2 p3 p4 p5 p6 p7
p8 p9 p10 p11 load_clear loadnot clearnot i2a i3a pass pass_acond i21 i31 bnnot i6not i7not )
        ( cinvert i0not i0 )
        ( cinvert i1not i1 )
        ( cinvert i2not i2 )
        ( cinvert i3not i3 )
        ( cinvert i4not i4 )
        ( cinvert i5not i5 )
;       Decode the 0 operand instructions
        ( cnand p0 i4 i3not i2not i1 i0 )                ; The CLEAR instruction is present
        ( cnand p1 i4 i3 i2 i1not i0not )                ; The SET instruction is present
        ( cnand p2 i5 i4not i3not i2not i1 i0 )          ; The NEG instruction is present
        ( cnand 0op p0 p1 p2 )                           ; Zero Operand instruction
;       Decode the 1 operand instructions
        ( cnand p3 i5not i4not i3not i2not i1not i0not )  ; The INC_A instruction
        ( cnand p4 i5not i4not i3 i2 i1not i0not )        ; The SHIFT_AC instruction
        ( cnand p5 i4 i3not i2not i1not i0not )           ; The NOT_A instruction
        ( cnand p6 i4 i3not i2 i1not i0 )                ; The NOT_B instruction
        ( cnand p7 i4 i3 i2not i1 i0not )                ; The PASS_B instruction
        ( cnand p8 i5 i4not i3not i2not i1not i0not )     ; The PASS_A instruction
        ( cnand p9 i5 i4not i3 i2 i1not i0not )           ; The SHIFT_A instruction
        ( cnand p10 i5 i4not i3 i2 i1 i0 )               ; The DEC_A instruction
        ( cnand sr i5 i4 i3 i2 i1 i0 )                   ; The SHIFTR_A instruction
        ( cnand 1op p3 p4 p5 p6 p7 p8 p9 p10 sr )        ; One Operand instruction
;       Decode the 2 operand instructions
        ( cnor 2op 0op 1op )                             ; Two Operand instruction
        ( cand5 p11 i4not i3 i2 i1 i0 )
        ( cand pass_acond i5not p11 )                    ; The PASS_ACOND instruction is present
        ( cand5 pass i4 i3 i2 i1 i0 )                    ; The PASS_AB instruction is present
```

```
        ( cor pass_ab pass load )                    ; Force PASS_AB if load or paqss is high
        ( cinvert shift_right sr )
;       Force the CLEAR instruction when the pass_acond instruction is present and bn = 0.
        ( cinvert bnnot bn )
        ( cinvert loadnot load )
        ( cnand clearnot bnnot pass_acond loadnot )
        ( cand i22 i2a clearnot )
        ( cand i32 i3a clearnot )
        ( cor load_clear load pass_acond )
;       Force the PASS_AB instruction
        ( cor i02 i0 load_clear )
        ( cor i12 i1 load_clear )
        ( cor i2a i2 load_clear )
        ( cor i3a i3 load_clear )
        ( cor i42 i4 load_clear )
;       Force the Route a -> c, b -> d on the PE core's output switch box, not on the RE
        ( cinvert i6not i6 )
        ( cinvert i7not i7 )
        ( cnor i61 i6not load )
        ( cnor i71 i7not load ))
;       The PE core
(macro core ( done 0op 1op 2op c7 c6 c5 c4 c3 c2 c1 c0 d7 d6 d5 d4 d3 d2 d1 d0 reset load req consumed
error a7 a6 a5 a4 a3 a2 a1 a0 b7 b6 b5 b4 b3 b2 b1 b0 )
;       a0-a7   = Input bus A
;       b0-b7   = Input bus B
;       reset   = Software reset of all registers
;       load    = We are programming the RIPP
;       req     = Request for computation to be done
;       error   = A synchronization error has been detected
;       consumed= The previously generated result has been used up
;       c0-c7   = Output bus C
;       d0-d7   = Output bus D
;       2op     = The IR contains a 2 operand instruction
;       1op     = The IR contains a 1 operand instruction
;       0op     = The IR contains a 0 operand instruction
;       done    = The core has completed its computation
(local int_done shift_right errorq p0 p1 p2 p3 p4 p5 p6 p7 i7 i6 i5 i4 i3 i2 i1 i0 i61 i71 i42 i32 i22 i12 i02
load_reg pass_ab r15 r14 r13 r12 r11 r10 r9 r8 r7 r6 r5 r4 r3 r2 r1 r0 f7 f6 f5 f4 f3 f2 f1 f0 cout eq gt lt p g c
fq7 fq6 fq5 fq4 fq3 fq2 fq1 fq0 s7 s6 s5 s4 s3 s2 s1 s0 gq pq ltq gtq eqq cq zq z )
;       load = 1 and b1 = 1; This means that we are loading instructions
        (cand load_reg load b1 )
```

```
        (register i7 i6 i5 i4 i3 i2 i1 i0 reset load_reg a7 a6 a5 a4 a3 a2 a1 a0)
;       Generate the PE internal control signals
        (cu i71 i61 i42 i32 i22 i12 i02 pass_ab shift_right 0op 1op 2op i7 i6 i5 i4 i3 i2 i1 i0 b0 load)
;       Hookup the ALU to the input.
        (alu int_done req f7 f6 f5 f4 f3 f2 f1 f0 cout z lt eq gt p g i5 i42 a7 a6 a5 a4 a3 a2 a1 a0 b7 b6 b5 b4
b3 b2 b1 b0 i32 i22 i12 i02)
;       Should we force B onto the status bus ?
        (mux s7 s6 s5 s4 s3 s2 s1 s0 pass_ab z cout g p error eq lt gt b7 b6 b5 b4 b3 b2 b1 b0)
;       Buffer the value in the Result and status buses only when
        ( cand done int_done consumed )
;       Latch the results at the right time
        (sregister fq7 fq6 fq5 fq4 fq3 fq2 fq1 fq0 reset shift_right done f7 f6 f5 f4 f3 f2 f1 f0)
        (register zq cq gq pq errorq eqq ltq gtq reset done s7 s6 s5 s4 s3 s2 s1 s0)
;       Route the Result and Status buses to the corresponding output
        (switch i71 i61 c7 c6 c5 c4 c3 c2 c1 c0 d7 d6 d5 d4 d3 d2 d1 d0 fq7 fq6 fq5 fq4 fq3 tq2 fq1 fq0 zq
cq gq pq errorq eqq ltq gtq ))
;       1 to 3 Demux
(macro demux41 ( d e f r0 r1 a )
;       d, e, f     = Outputs
;       a           = Input
;       r0, r1      = Control
( local r0n r1n )
        ( cinvert r0n r0 )
        ( cinvert r1n r1 )
        ( cand3 d r1n r0 a )
        ( cand3 e r1 r0n a )
        ( cand3 f r1 r0 a ))
;       1 to 3 Demux
(macro demux42 ( c e f r0 r1 a )
;       c, e, f     = Outputs
;       a           = Input
;       r0, r1      = Control
( local r0n r1n )
        ( cinvert r0n r0 )
        ( cinvert r1n r1 )
        ( cand3 c r1n r0n a )
        ( cand3 e r1 r0n a )
        ( cand3 f r1 r0 a ))
;       1 to 3 Demux
(macro demux43 ( c d f r0 r1 a )
;       c, d, f     = Outputs
```

```
;       a               = Input
;       r0, r1          = Control
( local r0n r1n )
        ( cinvert r0n r0 )
        ( cinvert r1n r1 )
        ( cand3 c r1n r0n a )
        ( cand3 d r1n r0 a )
        ( cand3 f r1 r0 a ))
;       1 to 3 Demux
(macro demux44 ( c d e r0 r1 a )
;       c, d, e   = Outputs
;       a               = Input
;       r0, r1          = Control
( local r0n r1n )
        ( cinvert r0n r0 )
        ( cinvert r1n r1 )
        ( cand3 c r1n r0n a )
        ( cand3 d r1n r0 a )
        ( cand3 e r1 r0n a ))
;       1 to 4 Demux
(macro demux4 ( c d e f r0 r1 a )
;       c, d, e, f = Outputs
;       a               = Input
;       r0, r1          = Control
( local r0n r1n )
        ( cinvert r0n r0 )
        ( cinvert r1n r1 )
        ( cand3 c r1n r0n a )
        ( cand3 d r1n r0 a )
        ( cand3 e r1 r0n a )
        ( cand3 f r1 r0 a ))
;       1 Bit Route Box
(macro bit_route ( a b c d dira dirb dirc dird r7 r6 r5 r4 r3 r2 r1 r0 )
;       a, b, c, d = Input/Output
;       dir*            = Direction for each bus a, b, c, d
;       r0 - r7   = Control the data flow groups
( local a1 a2 a3 b0 b2 b3 c0 c1 c3 d0 d1 d2 p0 p1 p2 p3)
        ( demux41 b0 c0 d0 r0 r1 a )
        ( demux42 a1 c1 d1 r2 r3 b )
        ( demux43 a2 b2 d2 r4 r5 c )
```

```
        ( demux44 a3 b3 c3 r6 r7 d )
        ( cor3 p0 a1 a2 a3 )
        ( ctgate a p0 dira )
        ( cor3 p1 b0 b2 b3 )
        ( ctgate b p1 dirb )
        ( cor3 p2 c0 c1 c3 )
        ( ctgate c p2 dirc )
        ( cor3 p3 d0 d1 d2 )
        ( ctgate d p3 dird ))
;       8 Bit Routing Element
(macro route ( a0 a1 a2 a3 a4 a5 a6 a7 b0 b1 b2 b3 b4 b5 b6 b7 c0 c1 c2 c3 c4 c5 c6 c7 d0 d1 d2 d3 d4 d5 d6
d7 load reset acka ackb ackc ackd reqa reqb reqc reqd)
;       a0 - a7         ,
;       b0 - b7         ,
;       c0 - c7         ,
;       d0 - d7         = Inputs/Outputs
;       ack*            = Sync signal
;       req*            = Sync signal
;       load            = Routing Element Instruction Register load signal
;       reset           = Signal to clear the RE IR
(local load_reg loadnot r0a r1a r2a r3a r4a r5a r6a r7a r0 r1 r2 r3
r4 r5 r6 r7 dira dirb dirc dird p0 p1)
;       load = 1 and b2 = 1; This means that we are loading the RE IR
        (cand load_reg load b2)
        (register r7a r6a r5a r4a r3a r2a r1a r0a reset load_reg a7 a6 a5 a4 a3 a2 a1 a0)
;       Make sure that the a-> c & b->d instruction ( 0xEE ) is enabled when load = 1
        (cinvert loadnot load)
        (cand r0 r0a loadnot)
        (cor r1 r1a load)
        (cor r2 r2a load)
        (cor r3 r3a load)
        (cand r4 r4a loadnot)
        (cor r5 r5a load)
        (cor r6 r6a load)
        (cor r7 r7a load)
;       The r1 = 0, and r0 = 0 then signal a is an output, else it is an input. When it is an output
;       it must be driven, else it must be left alone because it is being driven by some PE.
        ( cnor dira r1 r0 )
;       The r3 = 0, and r2 = 1 then signal b is an output, else it is an input. When it is an output
;       it must be driven, else it must be left alone because it is being driven by some PE.
        ( cinvert p0 r3 )
```

```
        ( cand dirb p0 r2 )
;       The r5 = 1, and r4 = 0 then signal c is an output, else it is an input. When it is an output it
;       must be driven, else it must be left alone because it is being driven by some PE.
        ( cinvert p1 r4 )
        ( cand dirc p1 r5 )
;       The r7 = 1, and r6 = 1 then signal d is an output, else it is an input. When it is an output it
;       must be driven, else it must be left alone because it is being driven by some PE.
        ( cand dird r7 r6 )
        ( bit_route acka ackb ackc ackd dira dirb dirc dird r7 r6 r5 r4 r3 r2 r1 r0 )
        ( bit_route reqa reqb reqc reqd dira dirb dirc dird r7 r6 r5 r4 r3 r2 r1 r0 )
        ( bit_route a0 b0 c0 d0 dira dirb dirc dird r7 r6 r5 r4 r3 r2 r1 r0 )
        ( bit_route a1 b1 c1 d1 dira dirb dirc dird r7 r6 r5 r4 r3 r2 r1 r0 )
        ( bit_route a2 b2 c2 d2 dira dirb dirc dird r7 r6 r5 r4 r3 r2 r1 r0 )
        ( bit_route a3 b3 c3 d3 dira dirb dirc dird r7 r6 r5 r4 r3 r2 r1 r0 )
        ( bit_route a4 b4 c4 d4 dira dirb dirc dird r7 r6 r5 r4 r3 r2 r1 r0 )
        ( bit_route a5 b5 c5 d5 dira dirb dirc dird r7 r6 r5 r4 r3 r2 r1 r0 )
        ( bit_route a6 b6 c6 d6 dira dirb dirc dird r7 r6 r5 r4 r3 r2 r1 r0 )
        ( bit_route a7 b7 c7 d7 dira dirb dirc dird r7 r6 r5 r4 r3 r2 r1 r0 ))
;       Sync Output PE
(macro syncout ( consumed reqc reqd ackc ackd done load reset )
;       ackc    = Computation on data in bus C has completed
;       ackd    = Computation on data in bus D has completed
;       done    = The core has completed its computation
;       load    = Programming the RIPP
;       consumed= The previously generated result has been used up
;       reqc    = Request for computation on output data in bus C
;       reqd    = Request for computation on output data in bus D
        ( rsff reqc done ackc load reset )
        ( rsff reqd done ackd load reset )
        ( cnor consumed reqc reqd ))
; Sync Input PE
(macro syncin (req error load reqa reqb 2op 1op 0op)
;       load    = Programming the RIPP
;       reqa    = Request for computation on data in bus A
;       reqb    = Request for computation on data in bus B
;       2op     = The IR contains a 2 operand instruction
;       1op     = The IR contains a 1 operand instruction
;       0op     = The IR contains a 0 operand instruction
;       req     = Start the next computation on the core
;       error   = A synchronization error has been detected
```

```
(local notload p0 p1 p2 p3 p4 p5 p6 p7 p8 p9)
        ( cand p0 reqa reqb )
        ( cnand p1 p0 2op )
        ( cinvert p2 p1 )           ; The cu signals a 2op and we have two incoming requests
        ( cor p3 reqa reqb )
        ( cnand p4 p3 p1 1op )      ; The cu signals a 1op and we have at least one request
        ( cinvert p5 p4 )
        ( cand3 p6 p4 p1 0op )      ; The cu signals a 0op and we have no reqa or reqb
        ( cor4 req p2 p5 p6 load ); It is OK to generate a req even if there is a sync error
        ( cinvert p7 2op )
        ( cinvert notload load )
        ( cand3 p8 p7 p0 notload); We have 2 reqs  but the cu does not signal a 2op instruction
        ( cand3 p9 p3 0op notload); We have 1 req, but the cu signals a 0op instruction
        ( cor error p8 p9 ))        ; There is a sync error unless we are loading
;       Put together the PE with a core and both sync in/out units.
(macro pe ( reset load a7 a6 a5 a4 a3 a2 a1 a0 b7 b6 b5 b4 b3 b2 b1 b0 c7 c6 c5 c4 c3 c2 c1 c0 d7 d6 d5 d4
d3 d2 d1 d0 reqa reqb reqc reqd acka ackb ackc ackd )
( local req error 2op 1op 0op done consumed )
        ( syncin req error load reqa reqb 2op 1op 0op )
        ( core done 0op 1op 2op c7 c6 c5 c4 c3 c2 c1 c0 d7 d6 d5 d4 d3 d2 d1 d0 reset load req consumed
error a7 a6 a5 a4 a3 a2 a1 a0 b7 b6 b5 b4 b3 b2 b1 b0 )
        ( syncout consumed reqc reqd ackc ackd done load reset )
        ( connect acka reqc)
        ( connect ackb reqd ))
```

APPENDIX D

Automatic Simulation Makefile

```
#
# Packages to use
#
DIR = src
SRC = ../src
TARGETS = alu alu4 alu4l bit_route core cu demux4 dff lacg mux mux2 register route rsff sbox
switch sync syncout syncin tgate sregister pe row

.SUFFIXES: .result .rnl .sim

.sim.rnl:
 presim $*.sim $@

.rnl.result:
        ./$* | rnl | grep MARCO | sed > $@ -e "1,1d" -e "s/MARCO/ /g"
        rm -f $* $*.rnl $*.sim $*.al

targets: $(TARGETS)

all:
        cd $(DIR); make all; cd ..;
bin:
        cd $(DIR); make bin; cd ..;

clean:
        cd $(DIR); make clean; cd ..;
        rm -f *.result *.rnl $(TARGETS) *.log *.out *.sim *.al

veryclean: clean
        cd $(SRC); make clean; cd ..;

$(TARGETS):
        cd $(SRC);      make clean $@;          cd ..;
        cd $(DIR);      make clean $@;          cd ..;
        mv $(SRC)/$@.sim $(SRC)/$@.al .
        make $@.rnl $@.result
```

APPENDIX E

Automatic Simulation Step Output

```
nu.mzelada % make pe
cd ../src;          make clean pe;          cd ..;
rm -f alu alu4 alu4l bit_route core cu demux4 dff lacg mux mux2 register route rsff sbox switch sync syncout syncin tgate sregister pe row *.sim *.al *.names *.nodes *.log *.rnl core
make pe.sim
netlist pe.net pe.sim -tcmos-pw
cd src;          make clean pe;          cd ..;
rm -f alu alu4 alu4l bit_route core cu demux4 dff lacg mux mux2 register route rsff sbox switch sync syncout syncin tgate sregister pe row *.o core
cc -w -I../../include -DRNL -o pe pe.c
install -c -s -m 0755 pe ..
mv ../src/pe.sim ../src/pe.al .
make pe.rnl pe.result
presim pe.sim pe.rnl
Version 4.2
1507 nodes; transistors: enh=1477 intrinsic=0 p-chan=1477 dep=0 low-power=0 pullup=0 resistor=0
Total transistors eliminated = 2954
./pe | rnl | grep MARCO | sed > pe.result -e "1,1d" -e "s/MARCO/ /g"
RNL Version 4.2
rm -f pe pe.rnl pe.sim pe.al
```

APPENDIX F

Partial Simulation Results

```
1   ctl=0b10 r=0x0 i=0x0 req=00XX ack=XX00 a=0x0 b=0x0 c=0x0 d=0x0
2   ctl=0b00 r=0x0 i=0x0 req=0b0000 ack=0b0000 a=0x0 b=0x0 c=0x0 d=0x0
3   ctl=0b01 r=0x0 i=0x29 req=0b0011 ack=0b0000 a=0x29 b=0x2 c=0x0 d=0x0
4   ctl=0b00 r=0x0 i=0x29 req=0b0011 ack=0b0000 a=0x29 b=0x2 c=0x0 d=0x0
5   ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b0000 a=0x0 b=0x0 c=0x0 d=0x0
6   ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b1111 a=0x0 b=0x0 c=0x0 d=0x82
7   ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b0000 a=0x0 b=0x0 c=0x0 d=0x82
8   ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x0 c=0x0 d=0x82
9   ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x0 c=0x0 d=0x82
10  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x0 c=0x0 d=0x82
11  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b0000 a=0x0 b=0x1 c=0x0 d=0x82
12  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b1111 a=0x0 b=0x1 c=0x1 d=0x2
13  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b0000 a=0x0 b=0x1 c=0x1 d=0x2
14  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x1 c=0x1 d=0x2
15  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x2 c=0x1 d=0x2
16  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x4 c=0x1 d=0x2
17  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b0000 a=0x0 b=0x2 c=0x1 d=0x2
18  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b1111 a=0x0 b=0x2 c=0x2 d=0x2
19  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b0000 a=0x0 b=0x2 c=0x2 d=0x2
20  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x2 c=0x2 d=0x2
21  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x4 c=0x2 d=0x2
22  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x8 c=0x2 d=0x2
23  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b0000 a=0x0 b=0x3 c=0x2 d=0x2
24  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b1111 a=0x0 b=0x3 c=0x3 d=0x2
25  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b0000 a=0x0 b=0x3 c=0x3 d=0x2
26  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x3 c=0x3 d=0x2
27  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x6 c=0x3 d=0x2
28  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0xc c=0x3 d=0x2
29  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b0000 a=0x0 b=0x4 c=0x3 d=0x2
30  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b1111 a=0x0 b=0x4 c=0x4 d=0x2
31  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b0000 a=0x0 b=0x4 c=0x4 d=0x2
32  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x4 c=0x4 d=0x2
33  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x8 c=0x4 d=0x2
34  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x10 c=0x4 d=0x2
35  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b0000 a=0x0 b=0x5 c=0x4 d=0x2
36  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b1111 a=0x0 b=0x5 c=0x5 d=0x2
37  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b0000 a=0x0 b=0x5 c=0x5 d=0x2
38  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x5 c=0x5 d=0x2
39  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0xa c=0x5 d=0x2
40  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x14 c=0x5 d=0x2
41  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b0000 a=0x0 b=0x6 c=0x5 d=0x2
42  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b1111 a=0x0 b=0x6 c=0x6 d=0x2
```

```
43  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b0000 a=0x0 b=0x6 c=0x6 d=0x2
44  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x6 c=0x6 d=0x2
45  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0xc c=0x6 d=0x2
46  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x18 c=0x6 d=0x2
47  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b0000 a=0x0 b=0x7 c=0x6 d=0x2
48  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b1111 a=0x0 b=0x7 c=0x7 d=0x2
49  ctl=0b00 r=0x0 i=0x0 req=0b1111 ack=0b0000 a=0x0 b=0x7 c=0x7 d=0x2
50  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x7 c=0x7 d=0x2
51  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0xe c=0x7 d=0x2
52  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x1c c=0x7 d=0x2
53  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b0000 a=0x1 b=0x0 c=0x7 d=0x2
54  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b1111 a=0x1 b=0x0 c=0x1 d=0x2
55  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b0000 a=0x1 b=0x0 c=0x1 d=0x2
56  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x0 c=0x1 d=0x2
57  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x0 c=0x1 d=0x2
58  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x0 c=0x1 d=0x2
59  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b0000 a=0x1 b=0x1 c=0x1 d=0x2
60  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b1111 a=0x1 b=0x1 c=0x2 d=0x2
61  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b0000 a=0x1 b=0x1 c=0x2 d=0x2
62  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x1 c=0x2 d=0x2
63  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x2 c=0x2 d=0x2
64  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x4 c=0x2 d=0x2
65  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b0000 a=0x1 b=0x2 c=0x2 d=0x2
66  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b1111 a=0x1 b=0x2 c=0x3 d=0x2
67  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b0000 a=0x1 b=0x2 c=0x3 d=0x2
68  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x2 c=0x3 d=0x2
69  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x4 c=0x3 d=0x2
70  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x8 c=0x3 d=0x2
71  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b0000 a=0x1 b=0x3 c=0x3 d=0x2
72  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b1111 a=0x1 b=0x3 c=0x4 d=0x2
73  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b0000 a=0x1 b=0x3 c=0x4 d=0x2
74  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x3 c=0x4 d=0x2
75  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x6 c=0x4 d=0x2
76  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0xc c=0x4 d=0x2
77  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b0000 a=0x1 b=0x4 c=0x4 d=0x2
78  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b1111 a=0x1 b=0x4 c=0x5 d=0x2
79  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b0000 a=0x1 b=0x4 c=0x5 d=0x2
80  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x4 c=0x5 d=0x2
81  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x8 c=0x5 d=0x2
82  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x10 c=0x5 d=0x2
83  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b0000 a=0x1 b=0x5 c=0x5 d=0x2
84  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b1111 a=0x1 b=0x5 c=0x6 d=0x2
85  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b0000 a=0x1 b=0x5 c=0x6 d=0x2
86  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x5 c=0x6 d=0x2
87  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0xa c=0x6 d=0x2
88  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x14 c=0x6 d=0x2
```

```
89  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b0000 a=0x1 b=0x6 c=0x6 d=0x2
90  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b1111 a=0x1 b=0x6 c=0x7 d=0x2
91  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b0000 a=0x1 b=0x6 c=0x7 d=0x2
92  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x6 c=0x7 d=0x2
93  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0xc c=0x7 d=0x2
94  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x18 c=0x7 d=0x2
95  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b0000 a=0x1 b=0x7 c=0x7 d=0x2
96  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b1111 a=0x1 b=0x7 c=0x8 d=0x2
97  ctl=0b00 r=0x0 i=0x1 req=0b1111 ack=0b0000 a=0x1 b=0x7 c=0x8 d=0x2
98  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x7 c=0x8 d=0x2
99  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0xe c=0x8 d=0x2
100 ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x1c c=0x8 d=0x2
101 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b0000 a=0x2 b=0x0 c=0x8 d=0x2
102 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b1111 a=0x2 b=0x0 c=0x2 d=0x2
103 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b0000 a=0x2 b=0x0 c=0x2 d=0x2
104 ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0x0 c=0x2 d=0x2
105 ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x0 c=0x2 d=0x2
106 ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x0 c=0x2 d=0x2
107 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b0000 a=0x2 b=0x1 c=0x2 d=0x2
108 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b1111 a=0x2 b=0x1 c=0x3 d=0x2
109 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b0000 a=0x2 b=0x1 c=0x3 d=0x2
110 ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0x1 c=0x3 d=0x2
111 ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x2 c=0x3 d=0x2
112 ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x4 c=0x3 d=0x2
113 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b0000 a=0x2 b=0x2 c=0x3 d=0x2
114 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b1111 a=0x2 b=0x2 c=0x4 d=0x2
115 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b0000 a=0x2 b=0x2 c=0x4 d=0x2
116 ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0x2 c=0x4 d=0x2
117 ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x4 c=0x4 d=0x2
118 ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x8 c=0x4 d=0x2
119 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b0000 a=0x2 b=0x3 c=0x4 d=0x2
120 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b1111 a=0x2 b=0x3 c=0x5 d=0x2
121 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b0000 a=0x2 b=0x3 c=0x5 d=0x2
122 ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0x3 c=0x5 d=0x2
123 ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x6 c=0x5 d=0x2
124 ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0xc c=0x5 d=0x2
125 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b0000 a=0x2 b=0x4 c=0x5 d=0x2
126 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b1111 a=0x2 b=0x4 c=0x6 d=0x2
127 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b0000 a=0x2 b=0x4 c=0x6 d=0x2
128 ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0x4 c=0x6 d=0x2
129 ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x8 c=0x6 d=0x2
130 ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x10 c=0x6 d=0x2
131 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b0000 a=0x2 b=0x5 c=0x6 d=0x2
132 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b1111 a=0x2 b=0x5 c=0x7 d=0x2
133 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b0000 a=0x2 b=0x5 c=0x7 d=0x2
134 ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0x5 c=0x7 d=0x2
```

```
135 ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0xa c=0x7 d=0x2
136 ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x14 c=0x7 d=0x2
137 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b0000 a=0x2 b=0x6 c=0x7 d=0x2
138 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b1111 a=0x2 b=0x6 c=0x8 d=0x2
139 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b0000 a=0x2 b=0x6 c=0x8 d=0x2
140 ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0x6 c=0x8 d=0x2
141 ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0xc c=0x8 d=0x2
142 ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x18 c=0x8 d=0x2
143 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b0000 a=0x2 b=0x7 c=0x8 d=0x2
144 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b1111 a=0x2 b=0x7 c=0x9 d=0x2
145 ctl=0b00 r=0x0 i=0x2 req=0b1111 ack=0b0000 a=0x2 b=0x7 c=0x9 d=0x2
146 ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0x7 c=0x9 d=0x2
147 ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0xe c=0x9 d=0x2
148 ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x1c c=0x9 d=0x2
149 ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b0000 a=0x3 b=0x0 c=0x9 d=0x2
150 ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b1111 a=0x3 b=0x0 c=0x3 d=0x2
151 ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b0000 a=0x3 b=0x0 c=0x3 d=0x2
152 ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0x0 c=0x3 d=0x2
153 ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x0 c=0x3 d=0x2
154 ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x0 c=0x3 d=0x2
155 ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b0000 a=0x3 b=0x1 c=0x3 d=0x2
156 ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b1111 a=0x3 b=0x1 c=0x4 d=0x2
157 ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b0000 a=0x3 b=0x1 c=0x4 d=0x2
158 ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0x1 c=0x4 d=0x2
159 ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x2 c=0x4 d=0x2
160 ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x4 c=0x4 d=0x2
161 ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b0000 a=0x3 b=0x2 c=0x4 d=0x2
162 ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b1111 a=0x3 b=0x2 c=0x5 d=0x2
163 ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b0000 a=0x3 b=0x2 c=0x5 d=0x2
164 ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0x2 c=0x5 d=0x2
165 ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x4 c=0x5 d=0x2
166 ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x8 c=0x5 d=0x2
167 ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b0000 a=0x3 b=0x3 c=0x5 d=0x2
168 ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b1111 a=0x3 b=0x3 c=0x6 d=0x2
169 ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b0000 a=0x3 b=0x3 c=0x6 d=0x2
170 ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0x3 c=0x6 d=0x2
171 ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x6 c=0x6 d=0x2
172 ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0xc c=0x6 d=0x2
173 ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b0000 a=0x3 b=0x4 c=0x6 d=0x2
174 ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b1111 a=0x3 b=0x4 c=0x7 d=0x2
175 ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b0000 a=0x3 b=0x4 c=0x7 d=0x2
176 ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0x4 c=0x7 d=0x2
177 ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x8 c=0x7 d=0x2
178 ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x10 c=0x7 d=0x2
179 ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b0000 a=0x3 b=0x5 c=0x7 d=0x2
180 ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b1111 a=0x3 b=0x5 c=0x8 d=0x2
```

```
181  ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b0000 a=0x3 b=0x5 c=0x8 d=0x2
182  ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0x5 c=0x8 d=0x2
183  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0xa c=0x8 d=0x2
184  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x14 c=0x8 d=0x2
185  ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b0000 a=0x3 b=0x6 c=0x8 d=0x2
186  ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b1111 a=0x3 b=0x6 c=0x9 d=0x2
187  ctl= 0b00 r=0x0 i=0x3 req=0b1111 ack=0b0000 a=0x3 b=0x6 c=0x9 d=0x2
188  ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0x6 c=0x9 d=0x2
189  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0xc c=0x9 d=0x2
190  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x18 c=0x9 d=0x2
191  ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b0000 a=0x3 b=0x7 c=0x9 d=0x2
192  ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b1111 a=0x3 b=0x7 c=0xa d=0x2
193  ctl=0b00 r=0x0 i=0x3 req=0b1111 ack=0b0000 a=0x3 b=0x7 c=0xa d=0x2
194  ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0x7 c=0xa d=0x2
195  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0xe c=0xa d=0x2
196  ctl=0b00 r=0x0 i=0x0 req=0b0011 ack=0b0000 a=0x0 b=0x1c c=0xa d=0x2
197  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b0000 a=0x4 b=0x0 c=0xa d=0x2
198  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b1111 a=0x4 b=0x0 c=0x4 d=0x2
199  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b0000 a=0x4 b=0x0 c=0x4 d=0x2
200  ctl=0b00 r=0x0 i=0x4 req=0b0011 ack=0b0000 a=0x4 b=0x0 c=0x4 d=0x2
201  ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0x0 c=0x4 d=0x2
202  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x0 c=0x4 d=0x2
203  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b0000 a=0x4 b=0x1 c=0x4 d=0x2
204  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b1111 a=0x4 b=0x1 c=0x5 d=0x2
205  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b0000 a=0x4 b=0x1 c=0x5 d=0x2
206  ctl=0b00 r=0x0 i=0x4 req=0b0011 ack=0b0000 a=0x4 b=0x1 c=0x5 d=0x2
207  ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0x2 c=0x5 d=0x2
208  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x4 c=0x5 d=0x2
209  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b0000 a=0x4 b=0x2 c=0x5 d=0x2
210  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b1111 a=0x4 b=0x2 c=0x6 d=0x2
211  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b0000 a=0x4 b=0x2 c=0x6 d=0x2
212  ctl=0b00 r=0x0 i=0x4 req=0b0011 ack=0b0000 a=0x4 b=0x2 c=0x6 d=0x2
213  ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0x4 c=0x6 d=0x2
214  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x8 c=0x6 d=0x2
215  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b0000 a=0x4 b=0x3 c=0x6 d=0x2
216  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b1111 a=0x4 b=0x3 c=0x7 d=0x2
217  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b0000 a=0x4 b=0x3 c=0x7 d=0x2
218  ctl=0b00 r=0x0 i=0x4 req=0b0011 ack=0b0000 a=0x4 b=0x3 c=0x7 d=0x2
219  ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0x6 c=0x7 d=0x2
220  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0xc c=0x7 d=0x2
221  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b0000 a=0x4 b=0x4 c=0x7 d=0x2
222  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b1111 a=0x4 b=0x4 c=0x8 d=0x2
223  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b0000 a=0x4 b=0x4 c=0x8 d=0x2
224  ctl=0b00 r=0x0 i=0x4 req=0b0011 ack=0b0000 a=0x4 b=0x4 c=0x8 d=0x2
225  ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0x8 c=0x8 d=0x2
226  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x10 c=0x8 d=0x2
```

```
227  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b0000 a=0x4 b=0x5 c=0x8 d=0x2
228  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b1111 a=0x4 b=0x5 c=0x9 d=0x2
229  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b0000 a=0x4 b=0x5 c=0x9 d=0x2
230  ctl=0b00 r=0x0 i=0x4 req=0b0011 ack=0b0000 a=0x4 b=0x5 c=0x9 d=0x2
231  ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0xa c=0x9 d=0x2
232  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x14 c=0x9 d=0x2
233  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b0000 a=0x4 b=0x6 c=0x9 d=0x2
234  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b1111 a=0x4 b=0x6 c=0xa d=0x2
235  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b0000 a=0x4 b=0x6 c=0xa d=0x2
236  ctl=0b00 r=0x0 i=0x4 req=0b0011 ack=0b0000 a=0x4 b=0x6 c=0xa d=0x2
237  ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0xc c=0xa d=0x2
238  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x18 c=0xa d=0x2
239  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b0000 a=0x4 b=0x7 c=0xa d=0x2
240  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b1111 a=0x4 b=0x7 c=0xb d=0x2
241  ctl=0b00 r=0x0 i=0x4 req=0b1111 ack=0b0000 a=0x4 b=0x7 c=0xb d=0x2
242  ctl=0b00 r=0x0 i=0x4 req=0b0011 ack=0b0000 a=0x4 b=0x7 c=0xb d=0x2
243  ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0xe c=0xb d=0x2
244  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x1c c=0xb d=0x2
245  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b0000 a=0x5 b=0x0 c=0xb d=0x2
246  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b1111 a=0x5 b=0x0 c=0x5 d=0x2
247  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b0000 a=0x5 b=0x0 c=0x5 d=0x2
248  ctl=0b00 r=0x0 i=0x5 req=0b0011 ack=0b0000 a=0x5 b=0x0 c=0x5 d=0x2
249  ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0x0 c=0x5 d=0x2
250  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x0 c=0x5 d=0x2
251  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b0000 a=0x5 b=0x1 c=0x5 d=0x2
252  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b1111 a=0x5 b=0x1 c=0x6 d=0x2
253  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b0000 a=0x5 b=0x1 c=0x6 d=0x2
254  ctl=0b00 r=0x0 i=0x5 req=0b0011 ack=0b0000 a=0x5 b=0x1 c=0x6 d=0x2
255  ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0x2 c=0x6 d=0x2
256  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x4 c=0x6 d=0x2
257  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b0000 a=0x5 b=0x2 c=0x6 d=0x2
258  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b1111 a=0x5 b=0x2 c=0x7 d=0x2
259  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b0000 a=0x5 b=0x2 c=0x7 d=0x2
260  ctl=0b00 r=0x0 i=0x5 req=0b0011 ack=0b0000 a=0x5 b=0x2 c=0x7 d=0x2
261  ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0x4 c=0x7 d=0x2
262  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x8 c=0x7 d=0x2
263  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b0000 a=0x5 b=0x3 c=0x7 d=0x2
264  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b1111 a=0x5 b=0x3 c=0x8 d=0x2
265  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b0000 a=0x5 b=0x3 c=0x8 d=0x2
266  ctl=0b00 r=0x0 i=0x5 req=0b0011 ack=0b0000 a=0x5 b=0x3 c=0x8 d=0x2
267  ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0x6 c=0x8 d=0x2
268  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0xc c=0x8 d=0x2
269  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b0000 a=0x5 b=0x4 c=0x8 d=0x2
270  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b1111 a=0x5 b=0x4 c=0x9 d=0x2
271  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b0000 a=0x5 b=0x4 c=0x9 d=0x2
272  ctl=0b00 r=0x0 i=0x5 req=0b0011 ack=0b0000 a=0x5 b=0x4 c=0x9 d=0x2
```

```
273  ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0x8 c=0x9 d=0x2
274  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x10 c=0x9 d=0x2
275  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b0000 a=0x5 b=0x5 c=0x9 d=0x2
276  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b1111 a=0x5 b=0x5 c=0xa d=0x2
277  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b0000 a=0x5 b=0x5 c=0xa d=0x2
278  ctl=0b00 r=0x0 i=0x5 req=0b0011 ack=0b0000 a=0x5 b=0x5 c=0xa d=0x2
279  ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0xa c=0xa d=0x2
280  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x14 c=0xa d=0x2
281  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b0000 a=0x5 b=0x6 c=0xa d=0x2
282  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b1111 a=0x5 b=0x6 c=0xb d=0x2
283  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b0000 a=0x5 b=0x6 c=0xb d=0x2
284  ctl=0b00 r=0x0 i=0x5 req=0b0011 ack=0b0000 a=0x5 b=0x6 c=0xb d=0x2
285  ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0xc c=0xb d=0x2
286  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x18 c=0xb d=0x2
287  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b0000 a=0x5 b=0x7 c=0xb d=0x2
288  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b1111 a=0x5 b=0x7 c=0xc d=0x2
289  ctl=0b00 r=0x0 i=0x5 req=0b1111 ack=0b0000 a=0x5 b=0x7 c=0xc d=0x2
290  ctl=0b00 r=0x0 i=0x5 req=0b0011 ack=0b0000 a=0x5 b=0x7 c=0xc d=0x2
291  ctl=0b00 r=0x0 i=0x2 req=0b0011 ack=0b0000 a=0x2 b=0xe c=0xc d=0x2
292  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x1c c=0xc d=0x2
293  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b0000 a=0x6 b=0x0 c=0xc d=0x2
294  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b1111 a=0x6 b=0x0 c=0x6 d=0x2
295  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b0000 a=0x6 b=0x0 c=0x6 d=0x2
296  ctl=0b00 r=0x0 i=0x6 req=0b0011 ack=0b0000 a=0x6 b=0x0 c=0x6 d=0x2
297  ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0x0 c=0x6 d=0x2
298  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x0 c=0x6 d=0x2
299  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b0000 a=0x6 b=0x1 c=0x6 d=0x2
300  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b1111 a=0x6 b=0x1 c=0x7 d=0x2
301  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b0000 a=0x6 b=0x1 c=0x7 d=0x2
302  ctl=0b00 r=0x0 i=0x6 req=0b0011 ack=0b0000 a=0x6 b=0x1 c=0x7 d=0x2
303  ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0x2 c=0x7 d=0x2
304  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x4 c=0x7 d=0x2
305  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b0000 a=0x6 b=0x2 c=0x7 d=0x2
306  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b1111 a=0x6 b=0x2 c=0x8 d=0x2
307  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b0000 a=0x6 b=0x2 c=0x8 d=0x2
308  ctl=0b00 r=0x0 i=0x6 req=0b0011 ack=0b0000 a=0x6 b=0x2 c=0x8 d=0x2
309  ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0x4 c=0x8 d=0x2
310  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x8 c=0x8 d=0x2
311  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b0000 a=0x6 b=0x3 c=0x8 d=0x2
312  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b1111 a=0x6 b=0x3 c=0x9 d=0x2
313  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b0000 a=0x6 b=0x3 c=0x9 d=0x2
314  ctl=0b00 r=0x0 i=0x6 req=0b0011 ack=0b0000 a=0x6 b=0x3 c=0x9 d=0x2
315  ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0x6 c=0x9 d=0x2
316  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0xc c=0x9 d=0x2
317  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b0000 a=0x6 b=0x4 c=0x9 d=0x2
318  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b1111 a=0x6 b=0x4 c=0xa d=0x2
```

```
319  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b0000 a=0x6 b=0x4 c=0xa d=0x2
320  ctl=0b00 r=0x0 i=0x6 req=0b0011 ack=0b0000 a=0x6 b=0x4 c=0xa d=0x2
321  ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0x8 c=0xa d=0x2
322  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x10 c=0xa d=0x2
323  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b0000 a=0x6 b=0x5 c=0xa d=0x2
324  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b1111 a=0x6 b=0x5 c=0xb d=0x2
325  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b0000 a=0x6 b=0x5 c=0xb d=0x2
326  ctl=0b00 r=0x0 i=0x6 req=0b0011 ack=0b0000 a=0x6 b=0x5 c=0xb d=0x2
327  ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0xa c=0xb d=0x2
328  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x14 c=0xb d=0x2
329  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b0000 a=0x6 b=0x6 c=0xb d=0x2
330  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b1111 a=0x6 b=0x6 c=0xc d=0x2
331  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b0000 a=0x6 b=0x6 c=0xc d=0x2
332  ctl=0b00 r=0x0 i=0x6 req=0b0011 ack=0b0000 a=0x6 b=0x6 c=0xc d=0x2
333  ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0xc c=0xc d=0x2
334  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x18 c=0xc d=0x2
335  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b0000 a=0x6 b=0x7 c=0xc d=0x2
336  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b1111 a=0x6 b=0x7 c=0xd d=0x2
337  ctl=0b00 r=0x0 i=0x6 req=0b1111 ack=0b0000 a=0x6 b=0x7 c=0xd d=0x2
338  ctl=0b00 r=0x0 i=0x6 req=0b0011 ack=0b0000 a=0x6 b=0x7 c=0xd d=0x2
339  ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0xe c=0xd d=0x2
340  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x1c c=0xd d=0x2
341  ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b0000 a=0x7 b=0x0 c=0xd d=0x2
342  ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b1111 a=0x7 b=0x0 c=0x7 d=0x2
343  ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b0000 a=0x7 b=0x0 c=0x7 d=0x2
344  ctl=0b00 r=0x0 i=0x7 req=0b0011 ack=0b0000 a=0x7 b=0x0 c=0x7 d=0x2
345  ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0x0 c=0x7 d=0x2
346  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x0 c=0x7 d=0x2
347  ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b0000 a=0x7 b=0x1 c=0x7 d=0x2
348  ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b1111 a=0x7 b=0x1 c=0x8 d=0x2
349  ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b0000 a=0x7 b=0x1 c=0x8 d=0x2
350  ctl=0b00 r=0x0 i=0x7 req=0b0011 ack=0b0000 a=0x7 b=0x1 c=0x8 d=0x2
351  ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0x2 c=0x8 d=0x2
352  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x4 c=0x8 d=0x2
353  ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b0000 a=0x7 b=0x2 c=0x8 d=0x2
354  ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b1111 a=0x7 b=0x2 c=0x9 d=0x2
355  ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b0000 a=0x7 b=0x2 c=0x9 d=0x2
356  ctl=0b00 r=0x0 i=0x7 req=0b0011 ack=0b0000 a=0x7 b=0x2 c=0x9 d=0x2
357  ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0x4 c=0x9 d=0x2
358  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x8 c=0x9 d=0x2
359  ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b0000 a=0x7 b=0x3 c=0x9 d=0x2
360  ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b1111 a=0x7 b=0x3 c=0xa d=0x2
361  ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b0000 a=0x7 b=0x3 c=0xa d=0x2
362  ctl=0b00 r=0x0 i=0x7 req=0b0011 ack=0b0000 a=0x7 b=0x3 c=0xa d=0x2
363  ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0x6 c=0xa d=0x2
364  ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0xc c=0xa d=0x2
```

```
365     ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b0000 a=0x7 b=0x4 c=0xa d=0x2
366     ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b1111 a=0x7 b=0x4 c=0xb d=0x2
367     ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b0000 a=0x7 b=0x4 c=0xb d=0x2
368     ctl=0b00 r=0x0 i=0x7 req=0b0011 ack=0b0000 a=0x7 b=0x4 c=0xb d=0x2
369     ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0x8 c=0xb d=0x2
370     ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x10 c=0xb d=0x2
371     ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b0000 a=0x7 b=0x5 c=0xb d=0x2
372     ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b1111 a=0x7 b=0x5 c=0xc d=0x2
373     ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b0000 a=0x7 b=0x5 c=0xc d=0x2
374     ctl=0b00 r=0x0 i=0x7 req=0b0011 ack=0b0000 a=0x7 b=0x5 c=0xc d=0x2
375     ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0xa c=0xc d=0x2
376     ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x14 c=0xc d=0x2
377     ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b0000 a=0x7 b=0x6 c=0xc d=0x2
378     ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b1111 a=0x7 b=0x6 c=0xd d=0x2
379     ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b0000 a=0x7 b=0x6 c=0xd d=0x2
380     ctl=0b00 r=0x0 i=0x7 req=0b0011 ack=0b0000 a=0x7 b=0x6 c=0xd d=0x2
381     ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0xc c=0xd d=0x2
382     ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x18 c=0xd d=0x2
383     ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b0000 a=0x7 b=0x7 c=0xd d=0x2
384     ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b1111 a=0x7 b=0x7 c=0xe d=0x2
385     ctl=0b00 r=0x0 i=0x7 req=0b1111 ack=0b0000 a=0x7 b=0x7 c=0xe d=0x2
386     ctl=0b00 r=0x0 i=0x7 req=0b0011 ack=0b0000 a=0x7 b=0x7 c=0xe d=0x2
387     ctl=0b00 r=0x0 i=0x3 req=0b0011 ack=0b0000 a=0x3 b=0xe c=0xe d=0x2
388     ctl=0b00 r=0x0 i=0x1 req=0b0011 ack=0b0000 a=0x1 b=0x1c c=0xe d=0x2
```