INFORMATION TO USERS

While the most advanced technology has been used to photograph and reproduce this manuscript, the quality of the reproduction is heavily dependent upon the quality of the material submitted. For example:

- Manuscript pages may have indistinct print. In such cases, the best available copy has been filmed.

- Manuscripts may not always be complete. In such cases, a note will indicate that it is not possible to obtain missing pages.

- Copyrighted material may have been removed from the manuscript. In such cases, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, and charts) are photographed by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is also filmed as one exposure and is available, for an additional charge, as a standard 35mm slide or as a 17”x 23” black and white photographic print.

Most photographs reproduce acceptably on positive microfilm or microfiche but lack the clarity on xerographic copies made from the microfilm. For an additional charge, 35mm slides of 3”x 9” black and white photographic prints are available for any photographs or illustrations that cannot be reproduced satisfactorily by xerography.
Ling, Yong-Long Calvin

HIERARCHICAL MULTIPROCESSOR ARCHITECTURE DESIGN IN VLSI FOR REAL-TIME ROBOTIC CONTROL APPLICATIONS

The Ohio State University Ph.D. 1986

University Microfilms International 300 N. Zeeb Road, Ann Arbor, MI 48106
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark √.

1. Glossy photographs or pages
2. Colored illustrations, paper or print
3. Photographs with dark background
4. Illustrations are poor copy
5. Pages with black marks, not original copy
6. Print shows through as there is text on both sides of page
7. Indistinct, broken or small print on several pages √
8. Print exceeds margin requirements
9. Tightly bound copy with print lost in spine
10. Computer printout pages with indistinct print
11. Page(s) lacking when material received, and not available from school or author.
12. Page(s) seem to be missing in numbering only as text follows.
13. Two pages numbered. Text follows.
14. Curling and wrinkled pages
15. Dissertation contains pages with print at a slant, filmed as received
16. Other

University Microfilms International
HIERARCHICAL MULTIPROCESSOR ARCHITECTURE
DESIGN IN VLSI FOR REAL-TIME ROBOTIC CONTROL
APPLICATIONS

A Dissertation
Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of the Ohio State University

by

Yong-Long Calvin Ling, B.S.Physics, M.S.E.E.

* * * * *
The Ohio State University
1986

Dissertation Committee:
Professor Karl W. Olson
Professor David E. Orin
Professor Steven Bibyk

Approved by:
Adviser
Department of Electrical Engineering
To my mother
ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Karl W. Olson, for his guidance and support throughout my dissertation work. I would also like to thank Professor David E. Orin for his support and many helpful comments. I appreciate Professor Stephen Bibyk for serving on my reading committee. Many thanks to Barbara Elberfeld and Blaine Lilly Jr. for their efforts to edit this dissertation. Kathryn Lilly and Jonathan Chao deserve special thanks for answering so many questions and for their works which made my research work so much easier.

I would like to thank my family for their encouragement and support throughout these years.

My wife, who has been so patient and has provided me with constant, loving support, deserves my most special thanks of all.

This dissertation research is sponsored by the National Science Foundation.
VITA

April 16, 1952 ....................... Born—Taiwan, Republic of China

1974 ............................... B.S.Physics, Fu-Jen Catholic University,
                              Taiwan, Republic of China

1979 ............................... M.S.E.E., The Ohio State University,
                              Columbus, Ohio

1979–1984 ......................... Project Engineer,
                              Municipal Electric Company, City of
                              Columbus, Ohio

1984 ............................... Graduate Student,
                              Department of Electrical Engineering,
                              The Ohio State University,
                              Columbus, Ohio
# TABLE OF CONTENTS

**ACKNOWLEDGEMENTS** iii

**VITA** iv

**LIST OF FIGURES** ix

**LIST OF TABLES** xvi

**I. INTRODUCTION** 1

1.1 Introduction .................................................. 1
1.2 Organization .................................................. 4

**II. SURVEY OF PREVIOUS WORK** 7

2.1 Introduction .................................................. 7
2.2 Parallel Algorithms and Parallel Architectures ........ 9
2.3 VLSI Computer Processor Architecture .................. 12
2.4 Multiprocessor Systems for Robotic Applications ....... 18
2.5 Special Purpose VLSI Processors for Real-time Robotic Control 19
2.6 Summary .................................................. 21

**III. LAYERED MULTIPROCESSOR SYSTEM ARCHITECTURE DESIGN FOR REAL-TIME ROBOTIC CONTROL APPLICATIONS** 22
VI. THE ROBOTIC VECTOR PROCESSOR DESIGN AND VLSI LAYOUT IN CMOS

6.1 Introduction ................................................................. 97
6.2 RVP Clocking Scheme .................................................. 97
6.3 RVP Control ............................................................... 98
6.4 Dual Port Register File ............................................... 98
6.5 Floating Point Adder ................................................... 104
6.6 Floating Point Multiplier ............................................. 114
6.7 Shift and Broadcast Network ...................................... 120
6.8 I/O Channel ............................................................... 123
6.9 VLSI Layout, Simulation, Testing of RVP ..................... 123
6.10 Summary ................................................................. 152

VII. PARALLEL IMPLEMENTATION OF THE ROBOTIC CONTROL APPLICATIONS

7.1 Introduction ............................................................... 154
7.2 Basic Floating Point Operations Implemented by RSP .... 155
7.3 Basic Vector Operations Implemented by RVP ............... 156
7.4 Implementation of Robotic Control With One RSP and One,
   Two, and Four RVPS ....................................................... 160
7.5 Multirate Implementation With Multiple RSPs and RVPS ... 196
7.6 Summary ................................................................. 206

VIII. SUMMARY AND CONCLUSIONS

8.1 Research Contributions .............................................. 214
8.2 Research Extensions ................................................ 218
REFERENCES

A. EQUATIONS FOR INVERSE PLANT PLUS JACOBIAN CONTROL ALGORITHM 225

B. CIRCUIT DESIGN OF THE ROBOTIC VECTOR PROCESSOR 232

C. ROBOTIC SCALAR PROCESSOR IMPLEMENTATION OF BASIC VECTOR OPERATIONS 258

D. RVP IMPLEMENTATION OF BASIC VECTOR OPERATIONS 262

E. RECIPROCAL GENERATION USING RSP 270
LIST OF FIGURES

1 Control block diagram of Inverse Plant Plus Jacobian Control with the computational blocks identified ........................................................ 27
2 Modified IEEE single precision floating point format .......................... 29
3 Layered multiprocessor architecture for real-time robotic control. . 36
4 Layer 1 - Floating Point Processor architecture for parallel/pipelined scalar computation. ................................................................. 37
5 Layer 2 - SIMD architecture for 3 × 1 vector operations. ......... 39
6 Layer 3 - Synchronized SIMD array architecture for the execution of tightly coupled computational tasks. ................................. 41
7 Layer 4 - Decentralized MIMD architecture with three nodes for real-time robotic control. ................................................................. 44
8 Robotic Vector Processor block diagram ................................................. 50
9 Floating Point Processor block diagram showing special addressable registers. ................................................................. 52
10 RVP data-path block diagram showing special addressable registers. 59
11 Robotic Vector Processor instruction set ................................................. 61
12 Robotic Vector Processor Pipeline ........................................................ 66
13 Robotic Scalar Processor block diagram ................................................. 72
14 Common Memory bus time-multiplexing scheme ............................ 73
15 Scalar Processor Unit block diagram. 76
16 SPU instruction set. 79
17 Scalar Processor Unit instruction formats and Processor Status Words. 81
18 Scalar Processor Unit register file overlapped multiwindow scheme. 85
19 Scalar Processor Unit pipeline and data path requirements. 87
20 Vector Sequencer instruction set. 88
21 RSP-to-RSP data packet communication. 91
22 Overlapped scalar, vector, and communication operations. 94
23 Proposed Robotic Scalar Processor floor plan. 96
24 RVP clock scheme. 99
25 RVP control signals. 100
26 RVP dual-read, single write register file. 102
27 RVP register file timing. 103
28 FPA block diagram. 106
29 Four-bit Manchester carry propagation chain with partial carry lookahead. 111
30 FPA timing signals. 113
31 FPM block diagram. 115
32 Carry-Save-adder multiplier. 117
33 FPM control timing diagram. 121
34 Shift and Broadcast Network block diagram. 122
35 I/O channel block diagram. 124
36 I/O channel control timing signals. 125
37 Standard cell template. 132
38 RVP floor plan. 136
FPA floor plan .............................................................. 137
FPA critical path .......................................................... 140
FPA test chip block diagram ......................................... 142
Photograph of FPA test chip ........................................... 143
FPA test program using Tektronix DAS 9100 .................. 145
FPM floor plan ............................................................. 146
FPM critical path .......................................................... 148
Timing delay of 31-bit Manchester partial carry lookahead adder,
alignment, and overflow/underflow control circuits ........... 149
Carry-Save-Adder array critical path ............................... 150
FPM test chip block diagram ......................................... 153
Task graph of Inverse Plant Plus Jacobian Control algorithm with
estimated execution times ............................................. 161
Architecture using one RSP and one RVP ....................... 164
Task graph of Inverse Plant Plus Jacobian Control algorithm with
one RSP and one RVP .................................................. 165
Execution timing of the one-RSP-one-RVP implementation .... 166
Task graph of Homogeneous matrix computation implemented by
one RVP ................................................................. 168
Task graph of Jacobian computation implemented by one RVP .... 169
Task graph of Direct Kinematics computation implemented by one
RVP ................................................................. 170
Task graph of Joint Acceleration computation implemented by one
RVP ................................................................. 172
Task graph of Inverse Dynamics forward recursion computation implemented by one RVP .................................................. 173
Task graph of Inverse Dynamics backward recursion computation implemented by one RVP ................................................. 174
Task graph of Inner Loop Control computation implemented by one RVP .................................................................................. 175
Task graph of Force Transform computation implemented by one RSP ..................................................................................... 177
Task graph of LU decomposition computation implemented by one RSP .................................................................................. 178
Task graph of forward and backward substitution using L and U matrices implemented by one RSP .................................................. 180
Task graph of Outer Loop Control computation implemented by one RSP ................................................................................ 181
Architecture using one RSP and two RVPs ................................................................................................................................. 182
Task graph of one-RSP-two-RVP implementation ......................................................................................................................... 184
Execution timing of the one-RSP-two-RVP architecture .................................................................................................................. 185
Task graph of Homogeneous matrix computation implemented by two RVPs .................................................................................. 187
Task graph of Jacobian computation implemented by two RVPs ...................................................................................................... 188
Task graph of Inverse Dynamics forward recursion computation implemented by two RVPs .......................................................... 190
Task graph of Inverse Dynamics backward recursion computation implemented by two RVPs ....................................................... 191
Architecture using one RSP and four RVPs .................................................................................................................................. 193
<table>
<thead>
<tr>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td>Task graph of Inverse Plant Plus Jacobian Control algorithm using one RSP and four RVPS.</td>
</tr>
<tr>
<td>73</td>
<td>Execution timing of the one-RSP-four-RVP architecture.</td>
</tr>
<tr>
<td>74</td>
<td>Task graph of Inverse Plant Plus Jacobian Control algorithm showing four feedback loops and estimated execution time.</td>
</tr>
<tr>
<td>75</td>
<td>Architecture for dualrate implementation using two RSPs and two RVPs.</td>
</tr>
<tr>
<td>76</td>
<td>Task graph of Inverse Plant Plus Jacobian Control algorithm using the two-RSP-two-RVP architecture.</td>
</tr>
<tr>
<td>77</td>
<td>Execution timing of the dualrate implementation of the two-RSP-two-RVP architecture.</td>
</tr>
<tr>
<td>78</td>
<td>Architecture for triplerate implementation using three RSPs and four RVPs.</td>
</tr>
<tr>
<td>79</td>
<td>Task graph of Inverse Plant Plus Jacobian Control algorithm using the three-RSP-four-RVP architecture.</td>
</tr>
<tr>
<td>80</td>
<td>Execution timing of the triplerate implementation of the three-RSP-four-RVP architecture.</td>
</tr>
<tr>
<td>81</td>
<td>Architecture for quadruplerate implementation using four RSPs and seven RVPs.</td>
</tr>
<tr>
<td>82</td>
<td>Task graph of Inverse Plant Plus Jacobian Control algorithm implementation using the four-RSP-seven-RVP architecture.</td>
</tr>
<tr>
<td>83</td>
<td>Execution timing of the quadruplerate implementation of the four-RSP-seven-RVP architecture.</td>
</tr>
<tr>
<td>84</td>
<td>FPA controller</td>
</tr>
<tr>
<td>85</td>
<td>FPA controller output waveforms</td>
</tr>
</tbody>
</table>
110 Program, execution timing, and reservation table for vector cross product. .................................................. 263
111 Program, execution timing, and reservation table for vector dot product. .................................................. 264
112 Program, execution timing, and reservation table for matrix vector multiplication. .................................. 265
113 Timing of overlapped vector operations with a previous V * V operation. ............................................... 266
114 Timing of overlapped vector operations with a previous V + V operation. ............................................... 267
115 Timing of overlapped vector operations with a previous V × V operation. ............................................... 268
116 Timing of overlapped vector operations with a previous MV or V · V operation. ..................................... 269
LIST OF TABLES

1 Computational requirements of Inverse Plant Plus Jacobian Control algorithm for a six-link robot manipulator. ................................ 31

2 Device parameters of the MOSIS 3 \( \mu \text{m} \) P-well CMOS fabrication process. .......................................................... 130

3 SPU execution time. ................................................................. 157

4 RVP execution time. .............................................................. 159

5 Performance comparison between various configurations. ........ 211
1.1 Introduction

The applications of robotic systems in the industrial world increased dramatically in the past decade. As a result of the automation revolution, robots are assuming more complex and demanding tasks. However, the potential of robotic systems is presently unrealized due to the limited computational power currently available. Real-time applications of robotic systems present more challenges to the control systems. The purpose of this dissertation research is to design a computer architecture that provides sufficient computing power to realize advanced robotic control systems in real-time.

The design of real-time robotic control systems involves several disciplines: the development of concurrent algorithms that lend themselves to parallel implementation on multiprocessor systems, the design of multiprocessor architectures that permits the concurrent processing of concurrent tasks, and circuit design that takes advantage of the state-of-the-art semiconductor technology. The designer of such systems requires a broad knowledge of, and experience with, all aspects of algorithms, software, hardware, performance evaluation, and computing alternatives.

The control schemes for robotic systems are complex and computationally intensive. The effectiveness of an algorithm depends greatly upon the
concurrent properties of the algorithm to permit concurrent execution. Few advanced control schemes for robotic systems have been developed to exploit computation concurrency. However, there is still a certain amount of parallelism embedded in these schemes, such as the parallelism between the computation of vector elements. Parallelism usually appears in several different forms of granularity. The first task to design an effective architecture is to identify these available parallelism. One robotic control algorithm, the Inverse Plant Plus Jacobian Control algorithm, developed by Orin [1] at The Ohio State University will be used as the target application for the architectures designed in this dissertation. The implementations of other robotic control applications, such as the Hybrid Position/Force Control algorithm for manipulators [2], may yield similar performance because of their similar computations.

Until this time, none of the multiprocessor systems implemented by commercially available microprocessors is able to meet the computation requirements for real-time control of advanced robotic control schemes. In Lilly’s thesis work [3], a multiprocessor system consisting of two Intel 86/30 Single Board Computers requires 194 ms to perform the computation of the Inverse Plant Plus Jacobian Control algorithm. Neither are the computational speeds of these processors fast enough, nor are they capable of handling the amount of data traffic involved in parallel processing. This has prompted the search for special computer architectures for real-time robotic control applications.

Research efforts in seeking for an efficient parallel architecture for real-time control of robotic systems have resulted in several special purpose computing structures. Turney and Mudge [4] have developed specifications for a single chip numerical processor intended for robotic control. Chao [5] has also proposed a VLSI Robotic Processor architecture specifically for robotic control applications. These
computing structures are arranged in parallel/pipelined configurations to perform special calculations. However, most of these special architectures concentrated mainly on the computational aspects of the robotic control systems. These special computing structures can function only as attached processors of general purpose computers. The interface between the general purpose computer and these special computing structures often became the bottleneck of these systems. In this dissertation, the design of a multiprocessor architecture which approaches the computation and communication problems in a layered fashion to avoid mismatches between them will be presented.

The advances in VLSI technology have changed the complexion of computer system design dramatically [6,7]. A state-of-the-art CMOS chip may include more than a million transistors with a gate delay of about a nanosecond. System architects are now able to implement several functional blocks into one single chip which were previously on separate chips. With the advances in computer-aided-design (CAD) software and the availability of VLSI fabrication services, custom chip design is becoming common practice. The recent trend toward “softened” hardware design has revolutionized the design of computer architectures. This dissertation will aim at implementing the architecture using several special VLSI chips.

Advances in VLSI technology have also changed some design considerations. The trade-off between speed and complexity is much more pronounced in VLSI design than in those designs implemented with discrete components. Furthermore, the performance parity between on-chip and off-chip communications greatly influences the design strategy. The UC Berkeley Reduced-Instruction-Set Computer (RISC) [8] project has demonstrated a significant performance gain by trading away infrequently used instructions for functional units having greater impact on the processor performance. The architecture designed in this dissertation will be
based upon this new design philosophy to better utilize valuable silicon area.

The objective of this dissertation research is to design a "layered multiproces­sor computer architecture using state-of-the-art VLSI technology to perform the control of robotic systems in real time. The research includes three major areas: the architectural design for the parallel processing of robotic computation, the mapping of the Inverse Plant Plus Jacobian Control algorithm onto the designed architecture, and the design and layout of the VLSI processors specified in the architecture.

The architectural design in this research follows a systematic approach of evaluating the concurrency and data dependencies embedded in the various levels of computation. Parallelism and pipelining [9] are employed at the appropriate levels to limit the adverse effects of design trade-offs. Designs are conducted in a layered fashion so that each layer of computing structure serves as the underlying structure for the next layer. Each layer in the computing hierarchy is responsible for its own targeted concurrency. The definition of these layers and the interfaces between them cleanly dissect the design tasks and thus allow us to clarify the complex nature of the architectural design. The layered approach produces a hierarchical architecture that addresses the various parallel processing issues separately, while employing them to operate coherently with other layers.

1.2 Organization

Chapter 2 includes a survey of previous work in the areas of parallel processing and parallel structures, VLSI computer architectures, multiprocessor systems for robotic control applications, and special VLSI processors for the real-time control of robotic systems.
Chapter 3 describes the design of the hierarchical system architecture. Architectural considerations for real-time robotic control are presented first, then the hierarchical multiprocessor architecture is presented in a layered fashion. The architecture hierarchy is divided into four layers: scalar, Single-Instruction-stream-Multiple-Data-stream (SIMD), synchronized SIMD array, and decentralized Multiple-Instruction-stream-Multiple-Data-stream (MIMD) [9]. Two VLSI processors, the Robotic Scalar Processor (RSP) and the Robotic Vector Processor (RVP), are then identified to be the building blocks of the layered architecture. The repeated use of these two VLSI processors permits the construction of various computing structures for different computational tasks.

The architecture of the RVP is described in Chapter 4. The RVP is an SIMD processor that includes three floating point processors, two interconnection networks, and two I/O channels. This processor is designed specifically for the computation of robotic algorithms, the majority of which are 3x1 vector operations. The floating point processor consists of a 32-bit floating point adder, a 32-bit floating point multiplier, and a large register file. Its architecture is designed to fully exploit the low level parallelism via pipelining, overlapped execution, and bus multiplexing. This single chip implementation greatly reduces the frequent data transfer between the processor elements of the SIMD structure. The control and communications design considerations for the formation of a synchronized SIMD array using a number of RVPs are also discussed.

Chapter 5 describes the architectural design of the Robotic Scalar Processor (RSP). The RSP is responsible for scalar computations, sequencing of vector operations, the I/O interface for the associated vector structure, and global communications. The Scalar Processor Unit (SPU) is a RISC processor with a register file, a 32-bit ALU, a floating point adder, and a floating point multiplier. The
design of a Common Memory (CM) architecture which integrates the scalar, vector, and I/O operations will be given.

Chapter 6 presents the design and layout of the RVP chip. The design of the RVP chip is partitioned into six major blocks: floating point adder, floating point multiplier, register file, shift and broadcast network, I/O channel, and processor controller. The floating point adder and multiplier have been laid-out with a CAD tool from UC Berkeley [10].

In Chapter 7, the implementation of the Inverse Plant Plus Jacobian Control algorithm is described. The control task is implemented by various structures to evaluate the performance of the hierarchical architecture. As the number of processors employed to speed-up the computation increases, so do the types of parallelism they exploit.

The last chapter summarizes the conclusions drawn from this dissertation research. Several possible research extensions are also suggested.

The equations for the Inverse Plant Plus Jacobian Control algorithm are presented in Appendix A. The detailed VLSI design of the RVP circuit is described in Appendix B. The execution sequence of the vector operations is included in Appendices C and D. Appendix E describes the reciprocal generation of 32-bit single precision floating point numbers.
CHAPTER II
SURVEY OF PREVIOUS WORK

2.1 Introduction

This chapter gives a brief overview of previous work involved in the areas of parallel/pipelined computer architectures, very-large-scale integrated circuit (VLSI) architectures, multiprocessor systems using commercially available VLSI processors for robotic applications, and special purpose VLSI processors for real-time robotic control.

The design of real-time robotic control systems involves three disciplines: development of concurrent algorithms that lend themselves to parallel implementation on multiprocessor systems, design of a multiprocessor architecture that permits concurrent processing of concurrent tasks, and circuit design that takes advantages of state-of-the-art semiconductor technology.

The control schemes for robotic systems are very complex and computationally intensive. They are difficult to implement in real-time using standard single processor computing structures [3]. Experience has shown that the real-time control of robotic system requires certain forms of parallel/pipelined computing structures which allow concurrent computation [3,11]. The effectiveness of an algorithm depends greatly on the concurrent properties of the algorithm to permit concurrent execution. Almost none of the advanced control schemes for robotic systems has been developed to exploit the computation concurrency. This makes the design
of effective multiprocessor systems very difficult because these architectures have
to fully exploit the parallelisms embedded in all granularity levels [12] in order to
obtain sufficient performance improvements.

Parallel and pipeline processing are the two approaches to achieve concurrent
computation in multiprocessor systems [9]. The system architect constantly faces
various factors which influence the design of the computing structure, often in a
contradictory way. The main task of the computer system architect is to weigh
these trade-offs and to make proper design decisions. The optimal computer ar-
chitecture is the one with the best performance-cost ratio. The second section of
the chapter reviews a number of papers dealing with the design issues of parallel/
pipelined computing structures.

The advances in the VLSI technology have changed the complexion of computer
system design. A state-of-the-art CMOS chip may include more than one million
transistors with a one nanosecond gate delay. System architects now are given
the opportunities to implement several function blocks with a single chip. The
performance parity between on-chip and off-chip communications presents a set of
new design criteria [6,7,13,14]. With the advances in the computer-aided-design
(CAD) software and the availability of VLSI fabrication services, custom chip
design is becoming common practice. This emergence of the trend of "softened"
hardware design has revolutionized the design of computer architectures. The third
section reviews some recent non-conventional VLSI processor architectures.

Section 2.4 reviews some multiprocessor system configurations for robotic ap-
lications using commercially available chips or boards. The design of these mul-
tiprocessor systems needs to deal with the issues of task partitioning, scheduling,
communications, memory access, and process synchronization. Research efforts
in searching for an efficient parallel architecture for real-time control of robotic
systems have resulted in several special purpose computing structures. These computing structures are arranged in parallel/pipelined configurations to perform special calculations or tasks. The fifth section reviews several special VLSI these computing structures.

2.2 Parallel Algorithms and Parallel Architectures

A parallel algorithm is viewed as a collection of independent task modules that can be executed in parallel and that communicate with each other during the execution of the algorithm. The most important task in designing computer structure for various applications is the development of a parallel algorithm for those applications. There is very close match between the parallel algorithms and the parallel architectures.

Kung [15,16] has identified three orthogonal dimensions of the space of parallel algorithms: concurrency control, module granularity, and communication geometry. Concurrency control enforces desired interactions among task modules so that the overall execution of the parallel algorithm will be correct. Module granularity of a parallel algorithm refers to the maximal amount of computation a typical task module can do before having to communicate with other modules. Communication geometry refers to the geometric layout of the network of interconnected task modules.

Kung then defined a space of parallel algorithms as the cross product

\{concurrency controls\} \times \{module granularities\} \times \{communication geometries\}.

The subspace \{MIMD, SIMD, systolic\} \times \{communication geometries\} represents most of the interesting and significant parallel algorithms where \{MIMD, SIMD, systolic\} are three particular positions in the space \{concurrency controls\} \times \{module granularities\}. The following subsections will discuss the various aspects
2.2.1 Parallel Processing

Parallel processing is an efficient form of information processing which emphasizes the exploitation of concurrent events in the computing process. Concurrency implies parallelism, simultaneity, and pipelining. These concurrent events are attainable in a computer system at various processing levels, namely, job or program level, task or procedure level, interinstruction level, and intrainstruction level. The highest job level is normally conducted algorithmically. The lowest intrainstruction level is often implemented directly by hardware implementation. As hardware cost declines and software cost increases, more and more hardware methods are replacing conventional software approaches. This trend is also supported by the increasing demand for a faster real-time, resource-sharing, and fault-tolerant computing environment.

The above characteristics suggest that parallel processing involves a combination of fields of studies. It requires a broad knowledge of and experience with all aspects of algorithms, languages, software, hardware, performance evaluation, and computing alternatives.

2.2.2 Parallel Computer Structures

Parallel computers are those systems which emphasize parallel processing. Basically, parallel computers can be divided into three architectural configurations [9]: Pipelined computers, Array processors, and Multiprocessor systems.

Pipelined computers perform overlapped computations with the same hardware to exploit temporal parallelism. Array processors execute the same instruction on an array of processors to achieve spatial parallelism. Multiprocessor systems achieve asynchronous parallelism through a set of interactive processors with
shared resources. These three structures are not mutually exclusive however and may be used together to achieve higher degree of multiplicity.

Pipelining is employed in almost all computer architectures because of its low hardware overhead and its ability to speed up the overall computation throughput by increasing the utilization of hardware resources. Pipelining may be exploited at various levels ranging from pipelining within an arithmetic unit to the pipelining of multiple processors.

Array processors are very effective for certain regularly structured computational tasks, such as vector operations. The linear speedup of array processors makes them most attractive. However, their hardware requirement is also linearly related to the amount of concurrency exploited. Furthermore, the performance of array processors frequently depends upon the performance of the communication structures they use, and the performance of these communication structures depends entirely upon the computational tasks executed on these processors. The lack of flexibility of array processors severely limits their applications.

Multiprocessor systems are more flexible than array processors; however, they also have a higher communication penalty. There are two architectural models for multiprocessor systems. One is the tightly coupled multiprocessor, where all processors communicate through a shared main memory. Another is the loosely coupled multiprocessor, where each processor has its own local memory and I/O channels. A special case of the loosely coupled multiprocessor systems is the distributed multiprocessor system. A distributed multiprocessing system is a collection of nearly autonomous processing nodes that communicate to realize a coherent computing system.

System architects need to carefully analyze the intended applications of the system, develop the evaluation criteria, weigh the various factors, and merge the
best combination of computing structures with the optimal set of trade-offs.

2.2.3 Algorithm Mapping

Algorithm mapping matches architecture topology with the algorithms so that parts of the algorithms correspond to certain architecture elements [17]. The computer structure should conform to the natural structure of the algorithm being performed. Algorithm mapping is an integral part of parallel processing. It may be implemented either before or after the system design. The earlier it is considered, the better the system will perform.

The objectives of algorithm mapping are:

- exploiting inherent parallelism of the algorithm at various levels,
- minimizing interprocessor communication,
- load balancing.

2.3 VLSI Computer Processor Architecture

As the result of rapid advances in VLSI technology, it is possible to implement circuits in excess of 100,000 transistors on a single chip. Functions formerly requiring separate chips can now be combined to yield superior performance. Although increasing the size or complexity of a digital circuit may lead to better system performance, it may also negatively affect performance as a result of longer wire delays, less driving power, and more circuit elements in the path of information flow. In VLSI systems the trade-off between speed and size/complexity of a circuit is more pronounced than it is in systems built from discrete components. Another important factor in VLSI system design is the large difference in available bandwidth between on-chip and off-chip communications.
Proper consideration of these trade-offs leads to hierarchically organized systems where the inner units are physically smaller and support the most frequently performed operations. The system architect has the important role of selecting the functions to be supported at the various levels of the system's hierarchy. This is particularly important in VLSI system design, where the spectrum of possible choices is wider and more continuous than it is in systems designed using discrete technology. Reduced-Instruction-Set-Computer (RISC) [18] architecture is designed with a new philosophy of complexity and speed trade-off.

Now that VLSI has effectively put a computer on a chip, designers are turning to the problem of building communication networks to link many processors to a single large system. Organizing large numbers of processing nodes, resolving communications-traffic snarls and dividing up sequential software for a well-balanced distribution of tasks are some of the problems posed by large networks of VLSI processors. The key to a successful VLSI system hinges upon its communication design. The traditional von Neumann bus-oriented interconnection has been replaced by a more general network topology[19,20]. For example, the Transputer [21,22] is a VLSI processor designed specifically to be integrated with large number of other Transputers to obtain performance comparable to supercomputers.

Computer architectures are generally characterized by their communication structure. A single ideal architecture for all programming problems may never crystallize from the profusion of geometric forms that are being tried. Instead, many now believe all that can be realized is sets of VLSI building blocks which can be assembled into architectures suitable for each generic class of problem. The Systolic Array [23] is an architecture that attempts to achieve computing performance through the arrangement of many VLSI processors into various forms of computing structures.
The following subsections will briefly introduce these three VLSI architectures.

### 2.3.1 Reduced Instruction Set Computer (RISC)

The philosophy behind RISC is to replace the traditional microcode store with a high speed cache memory. This approach eliminates the need to interpret a large instruction set with sequence of small instructions, thus speeding up execution time. The resulting system looks like a microcoded machine, except that the microcode instructions are closer to high-level language statements. It has been coined the term “millicode.” [24] Millicode is stored in main memory until required by the application. High level language statements correspond to millicode subroutines.

RISC’s single biggest impact may be a philosophical one: RISC technology challenges the trend toward increasingly complex processor architectures as the only path to improving computational performance. Present computer architecture represents an accumulation of ad hoc design decisions over the years. Statistical studies have shown a highly uneven use of instructions, suggesting that hardware costs needed to support seldom-used instructions might not be justified. Berkeley’s RISC project set out to build a 32-bit microprocessor that would ease VLSI design problems. The project also attempted to map the C language more efficiently into the hardware.

The Berkeley RISC project has demonstrated the viability of general purpose computers with simple instruction sets. It has revealed the non-optimal utilization of silicon resources in most contemporary single-chip processors, due to the increased complexity of their instruction sets.

There are three considerations in the design of a computer architecture:
1. Complex instructions used infrequently by actual programs have adverse effects on the overall system performance.

2. Operand accesses are so frequent that there should be better support for these accesses than is available in traditional architectures.

3. Simplified architecture is important in a field of rapidly changing technology because it leads to a short design time, thus allowing quick exploitation of the new technology.

From these considerations the Berkeley RISC architecture was derived. It specified a general purpose processor with simple instructions and with many registers organized in multiple register banks. They first identified the most necessary and frequent operations, then the data paths and timing for their execution were identified. At last, other frequent operations, which could also fit into that datapath and timing, were included into the instruction set.

Berkeley RISC Architecture was designed with the goal of allowing the integration of fast access units to operands and instructions, while still supporting the high-level performance execution of the simple operations required during most part of general-purpose computations.

The Berkeley RISC architecture, is register oriented. It has simple instructions, a simple and orthogonal instruction-format, and a regular timing model that fits all instructions. Fast operand accesses are supported by a large register-file with multiple overlapping window, where scalar arguments and local variables of procedures are allocated by default until the window is filled. Pipelining is utilized to keep operational units busy executing primitive operations selected by the compiler and optimizer. The combined effect of the 138 registers, organized in 8
windows, and of the 3-stage pipeline yields a machine of significantly higher performance than other commercial processors built in comparable technologies but with complex instructions and non-optimal use of registers. The size of programs compiled for RISC are not much larger than when they are compiled for other architectures, even though only simple instructions with a simple but code-inefficient format are available to the compiler.

The change toward instruction sets of increasing complexity has resulted in radical changes in the allocation of chip area to the various CPU functions. The control section of RISC II occupies only 10 percent of the area, in contrast with 50 percent for other Complicated-Instruction-Set Computers (CISC). Scarce silicon resources are thus freed and used more effectively for the implementation of the large register file. The simplicity of the processor contributes significantly to its speed by reducing the number of gate-delays in the critical path and also by reducing the physical size, and thus the parasitic capacitances, of the circuit elements. Designing, laying-out, and debugging the simple RISC II processor requires about five times less human effort than is normally required for other microprocessors.

2.3.2 Transputer

Today, a single wafer of silicon can contain hundreds of conventional microprocessors. To exploit this potential it will be necessary to build systems with a much higher degree of concurrency than is now possible. The Transputer is designed as a programmable component to implement such systems. The term "Transputer", which is derived from "transistor" and 'computer', reflects this new device's ability to be used as a system building block [22].

The Transputer is a RISC with high speed communication port on each of the four sides of the chip making it easy to link the chips together in processing...
arrays. The Transputer was designed for scalable parallel processing architectures. The Transputer was designed specifically to run with the Occam language, which itself is intended for parallel processing language.

The power of the Transputer is that it creates a new level of abstraction. Just as the use of logic gates and Boolean algebra provides the design methodology for present electronic systems, so the Transputer together with the formal rules of Occam, provides the design methodology for future concurrent systems.

Occam [25] captures the hierarchical structure of a system by allowing an interconnected set of processes to be seen from the outside as a single process. Those individual processes operate independently, for the most part, and the programmer need therefore be concerned only with a small and manageable set of processes at any level of detail. A key point of Occam is that the program is designed for parallelism, not for a specific number of processors, therefore, computational power can be expanded without altering the design again.

2.3.3 Systolic Array

The cost of communications in VLSI systems dominates the cost of computation. Properly designed parallel structures that need to communicate with only their nearest neighbors will gain the most from VLSI. Precious time is lost when modules that are far apart must communicate.

The systolic architecture, developed by Kung [23,26], consists of a set of interconnected cells, each capable of performing some simple operation. The simple and regular communication structure of systolic array has substantial advantages over complicated ones in design and implementation. Cells in a systolic system are typically interconnected to form a systolic array or a systolic tree. Information in a systolic system flows between cells in a pipelined fashion, and communications
with the outside world occurs only at the 'boundary' cells.

VLSI systolic arrays can assume many different structures for different compute-bound algorithms [27,28], such as signal processing and matrix arithmetic. The mapping of the algorithm onto a suitable configuration is most important. The mapping determines the execution of the algorithm and its performance. The major problem with a systolic array is still in its I/O barrier. The globally structured systolic array can speed-up computations only if the I/O bandwidth is high.

2.4 Multiprocessor Systems for Robotic Applications

Robotic control applications are computationally intensive. In order to achieve real-time control of a robotic system, several multiprocessor systems have been implemented at the Ohio State University.

Barrientos and Klein [29] have implemented a multimicroprocessor-based controller for a monopod vehicle using a structured design approach. Based upon the data-flow design method, they decomposed the control task into a series of related modules. The control kernel function includes four sequential modules: position estimation, motion planning, Jacobian control calculation, and servo level control. They have also integrated the system testability into their design. The results have shown good velocity and position tracking performance.

McGhee, Orin, Pugh and Patterson [30] have proposed a hierarchical control scheme for the ASV-84 walking machine. The hierarchical scheme decomposed the entire control so that different levels of control may be implemented by separate modules. The hierarchical scheme also allows higher levels of control, such as vision or other sensing functions, and may be added in a modular fashion without affecting the existing modules. The on-board computer system includes a guidance computer, a coordination computer, a cockpit computer and six leg computers.
Ozguner and Kao [31] have designed a fault tolerant multiprocessor system, the Selective Redundant Multiprocessor (SRMP), for the robotic control of a six legged robot. The SRMP consists of four processors connected through four busses with a fifth processor simulating the hexapod walking machine. The SRMP allows two or three processors to execute tasks in redundant configurations. Both hardware and software methods are employed to perform fault detection and correction; the comparisons and votings are performed by the software while the hardware comparators are used to speed-up the operations.

Lilly [3] has investigated the multiprocessor implementation of a real-time control system for robot manipulators. The control algorithm used in her research, the Inverse Plant Plus Jacobian Control algorithm, was developed by Orin [1]. The control algorithm is decomposed into eleven modules for multiprocessor implementation. The control scheme was implemented on a multiprocessor system of several Intel 86/30 single board computers. She has implemented the control algorithm on one and two processors, and obtained a speed-up of 1.37. She has also implemented the multirate control scheme to provide faster computation to more critical control loops such as the inner loop of the Jacobian control. The inner loop computation requires 6.35 ms while the other loops execute at rate ten times slower.

2.5 Special Purpose VLSI Processors for Real-time Robotic Control

All of the multiprocessor system implementations presented in the previous subsection use commercially available microprocessors. These commercial microprocessors present two problems. First, the floating point computation speeds of these microprocessors are too slow. Second, the communication structures of these multiprocessor systems do not provide sufficient bandwidth for effective
multiprocessing. For example, the inner loop in Lilly's multirate implementation requires 6.35 ms which is not fast enough to meet the real-time control requirement. Special VLSI processors for robotic control applications have been investigated to obtain sufficient computation performance.

Turney and Mudge [4] have developed specifications for a single chip numerical processor intended for robotic control applications. This processor was designed to function as an attached processor for a general purpose minicomputer. They proposed to include a register file, a floating point adder, a floating point multiplier, a set of I/O buffers, and on-chip program memory. The total number of transistors was estimated to be 150,000 for NMOS fabrication.

Both the floating point adder and multiplier had a three-stage pipeline with a cycle time of 500 ns. The two arithmetic units were connected to the register file by a two-bus structure. The instruction set included instructions for arithmetic operations, data transfer, and control branching. Programming of this processor was essentially implemented at the microcode level.

The objective of the numerical processor is to perform computationally intensive tasks such as the real-time robotic control. A simulation using the gross motion control as the benchmark showed an average of 2.93 MFLOPS, which represents 73 percent of its maximum speed.

Chao [5] proposed a VLSI Robotic Processor specifically for the computation of robotic control applications. The processor includes a floating point adder, a floating point multiplier, a register file, four I/O ports, and a control sequencer with a large on-chip program memory.

The arithmetic units also use three-stage pipelines. A three-bus interconnection structure transfers data between the arithmetic units, the register file, and the I/O ports. Programming of the robotic processor was implemented at the
microcode level.

The robotic processor designed by Chao was intended to form special computing structures for various computation tasks. These robotic processors were to be connected in parallel or in a pipeline to exploit the embedded concurrency of these tasks. He studied three applications in detail: the Jacobian, Inverse Jacobian, and Inverse Dynamics algorithms. Several different structures were studied for each application and evaluation of their performance were presented.

The robotic processor was to operate at 1 Mhz rate while the system clock rate was 16 Mhz. Chao concluded that all of the three tasks could be computed in 1 ms or less.

Both the numerical processor and the robotic processor have two major constraints: on-chip program memory limits the complexity of the applications and the I/O communication bottleneck to the external system.

2.6 Summary

This chapter has presented a brief overview of the previous works in the areas of multiprocessor system, VLSI processors, multiprocessor systems for robotic control, and special VLSI processors for robotic control.

The remaining chapters of this dissertation will present a hierarchical multiprocessor architecture based upon two VLSI processors to implement the Inverse Plant Plus Jacobian Control algorithm.
CHAPTER III

LAYERED MULTIPROCESSOR SYSTEM ARCHITECTURE
DESIGN FOR REAL-TIME ROBOTIC CONTROL APPLICATIONS

3.1 Introduction

An ideal architecture should be efficient, flexible, and cost effective. However, these objectives are often contradictory and dependent upon both the target applications and the technology used. The system designer has to weigh each objective carefully throughout the course of the architectural design.

The performance of any architecture can only be measured within the context of its application. A good system architecture design depends on well-conceived design criteria based upon the constraints and requirements of these applications. Therefore, the first step of system architecture design is to analyze the applications.

Real-time robotic control is the target application of the computer system to be designed. Section 3.2 presents the architectural design considerations for the real-time robotic control applications. A number of issues, such as the types of operands used, operations performed, sequence of execution, and communication requirements will be evaluated. The findings in Section 3.2 will reveal the computation and communications requirements as well as the potential parallelism embedded in the application.

Section 3.3 describes the layered multiprocessor architecture for the real-time robotic control applications based upon the findings in Section 3.2. This section
will describe the architecture from the global perspective. It defines the function of each layer, the types of concurrency targeted, and the interfaces among the layers.

Since the architecture has to be realized by VLSI chips, the implication of VLSI implementation is carefully evaluated in the design of every stage. The outcome is an architecture built upon two types of VLSI processors. Both processors will be used repeatedly to construct various computing structures matching their respective applications. This restructurable architecture design approach allows the user to tailor the architectures according to the needs of the applications, and achieve the objectives of speed and flexibility at reasonably low costs. The details of the architecture of the two VLSI chips will be discussed in the following two chapters.

3.2 Architectural Considerations for Real-Time Robotic Control

This section will discuss some of the considerations involved in the architectural design for real-time Inverse Plant Plus Jacobian Robotic Control. The purpose of the analysis is to obtain a thorough understanding of the performance requirements, constraints, and characteristics of the application. The findings in this section will serve as guidelines during the course of the architectural design.

The real-time performance requirement is very stringent and often requires a special computing structure. The design of a special computing structure usually requires more radical trade-offs in order to achieve the desired performance at reasonable cost. Subsection 3.2.1 presents some of the characteristics of the real-time control environment that lend themselves to a simplified computing structure.

Section 3.2.2 presents an analysis of the computation requirements for the Inverse Plant Plus Jacobian Control algorithm. Computational requirements, such
as the operands used, operations performed, and execution sequencing, determine the required functional units and the data bus structure supporting these units.

Data communications plays a major role in the choice of multiprocessor system because it often becomes the main obstacle to performance improvement as the system employs more processor elements. The bandwidth disparity between on-chip and off-chip data communications further complicates the issue. Subsection 3.2.3 will analyze the data communications requirement in the real-time robotic control environment. It influences the choice of architecture layers and the interface between them. It also plays a major role in the definition of the VLSI chips.

3.2.1 Real-Time Control Environment

Real-time control applications usually demand a great amount of computing power to provide prompt response. In the case of robotic control, the control system usually has less than a millisecond to compute a large number of vector operations. However, there are several characteristics of the real-time control environment that allow the designer to simplify the architectural design [32].

Regularity is most evident in the real-time control environment. The control system is to regularly sense the joint angles and rates from the robot and to respond with actuator signals that control the robot. This cyclic nature due to the regular sensing presents a static environment, with exception of some high level control schemes, to the control system. All tasks to be performed by the controller are known at the time of system configuration. This presents a set of clear objectives that allows the designer to make proper trade-offs during the course of system design.

Modularity is also quite apparent as shown in Figure 1 Each control module is very computationally intensive with highly localized data references, both
spatially and temporally. This presents the potential for parallel processing, both multiprocessing and pipelining.

Multiple unidirectional paths often can be found in real-time control systems. Every control system actually consists of several interconnected loops with each loop propagating the impact of data coming into the system, both command input or plant feedback. Each computational block is like a filter with data coming into the block and the result going out of it after some time delay. The whole computer system is like a group of filters with several data paths flowing through them.

Data-Driven [12] execution is another distinctive feature. Each computational block cycles through three stages: waiting for the input data, performing the computation, and sending the result to the next block or plant. All of the activities occurring in the system are triggered by the arrival of the input data. Tying the process scheduling with the data communications results in a decentralized data-driven multiprocessor system.

These distinctive characteristics of the real-time control applications will be exploited to the full extent during the architecture design to obtain the most cost effective parallel computing structure.

3.2.2 Analysis of Robotic Control Computation Requirements

The Inverse Plant Plus Jacobian Control Algorithm developed by Orin [1] will be used as the target application for the architecture developed. Figure 1 is the control block diagram of the Inverse Plant Plus Jacobian Control with the computational blocks identified [3].

Jacobian Control performs the robotic control based on joint rates rather than positions. If servoing and motion planning are accomplished in cartesian, workspace coordinates, it is necessary to perform the transformation between
gripper rates and joint rates. This transformation is the inverse Jacobian. Jacobian Control also includes position and force feedback, all at the gripper. It consists of two feedback loops, the outer loop and the inner loop.

Jacobian Control is based upon the kinematic properties of the mechanism; however, it is basically linear in nature and in most part does not account for the dynamic properties of a robotic mechanism. For this purpose, the Inverse Plant feedforward controller has been proposed, in which it is assumed that the desired position, rate, and acceleration of a mechanism are given and that the actuator torques are to be determined.

The combination of the Inverse Plant for feedforward control and the Jacobian Control for feedback has excellent potential for fast, accurate, and adaptive control.

This subsection will look into the computational requirements of this control algorithm. The main vehicle for a qualitative and quantitative understanding of the nature of computation is the measurement of the important properties of the application. These include the operands used, operations performed, and execution sequencing.

Operands Used

The type, size, and structure of the operands determine the types of arithmetic units to be implemented; the nature of their usage determines the storage organization and addressing modes.

The wide dynamic range of the robotic control variables requires the use of floating point representation. Most control variables appear in the form of $3 \times 1$ vectors which favor a three-bank memory structure. The number of variables involved in the robotic control computation is in the hundreds. This is not
Figure 1: Control block diagram of Inverse Plant Plus Jacobian Control with the computational blocks identified.
inordinately large. However, it is large enough to warrant careful consideration for the determination of the size of on-chip memory.

A 32-bit single precision floating point format is used to represent the control variables. A 64-bit double precision floating point format is not considered because it is very costly in terms of memory size, communications bandwidth, and computation time. The 24-bit significand of the single precision format is quite sufficient to support the resolution of the A/D and D/A converters used in the robotic control.

Figure 2 is the 32-bit single precision floating point format to be used in the floating point processor design. This is a modified IEEE standard single precision floating point format [33]. This format made up of a sign bit (S), an 8-bit biased exponent (E), and a 23-bit significand (F). There is an implicit bit, which is always a one, as the most significant bit of the mantissa. This makes the total number of bits of the mantissa 24.

The difference between the modified 32-bit floating point format and the IEEE standard floating point format lies in the way overflow and underflow numbers are handled. Whenever an underflow condition occurs, it is represented as $\pm 1.0 \times (2^{-127})$. On the other hand, all overflow numbers are represented as $\pm 1.1111\ldots11 \times (2^{128})$. There are no special reserved operands like those in the standard IEEE format.

The clamping of overflow/underflow numbers is very similar to the clamping of a D/A converter on an out-of-range input. This is necessary because in a real-time control environment the control system has to continue execution under the underflow or overflow circumstances. Continued execution with the clamped value may still yield sufficient accuracy. Furthermore, all physical data either coming into or going out of the control system are approximated values in the first place.
Integers are necessary to support the computation and logical operations required for complex program structures. Integers provide the control system with the capability to deal with more complex applications. A 32-bit integer format is used so that the transfer of integer numbers can be treated in the same way as the single precision floating point numbers.

**Operations Performed**

The type and frequency of operations determines the required operational units and the bus structure supporting these units. Table 1 summarizes an analysis of the static computation requirements of the Inverse Plant Plus Jacobian Control algorithm for a six-link robot manipulator. All vectors shown in this table are 3 \times 1 vector, and all matrices are 3 \times 3 matrices. The leftmost column shows the computational modules of this robotic control scheme. The Homogeneous Matrix module is not shown in the control block diagram, but is embedded in the modules using \mathbf{q}. Also, \mathbf{J}^{-1} block in Figure 1 actually consists of three modules listed in the
The module, Solution for $\ddot{q}_c$, is a part of the Joint Acceleration computational block.

Table 1 clearly shows that the majority of computations are vector additions, vector cross products, and matrix-vector multiplications. The success of the system depends very much on the performance of these operations. Note that modules 1 through 8 contain only vector operations.

Each vector cross product or matrix-vector multiplication operation consists of several vector multiplication and addition operations. A vector cross product requires two vector multiplications and a vector addition. A matrix-vector multiplication requires three vector multiplications and two vector additions. A vector dot product operation also uses three vector multiplications and two vector additions. The number of vector computations represented in terms of vector multiplication and addition is listed in the two columns on the left side of Table 1. It shows that the number of vector multiplications is very similar to that of the vector additions. This indicates that the computation bandwidth of the floating point multiplier and adder should be comparable.

Since the scalar floating point operations are required in the Inverse Jacobian computation the system has to include the scalar floating point processor. In addition to computing the Inverse Jacobian, the scalar processor can share the computational burden of the vector processor at a slower speed. Proper arrangement to coordinate the vector and scalar operations is crucial to achieve higher system throughput.

Common CPU functions such as the integer ALU functions, control transfer operations, memory access, and IO operations are necessary to support the software for the multiprocessor system.
Table 1: Computational requirements of Inverse Plant Plus Jacobian Control algorithm for a six-link robot manipulator.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>V * V</td>
<td>V + V</td>
<td>V x V</td>
</tr>
<tr>
<td>1. Homogeneous Matrix</td>
<td>64</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>2. Direct Kinematics</td>
<td>7</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>3. Force Transform</td>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>4. Outer Loop Control</td>
<td>4</td>
<td>13</td>
<td>9</td>
</tr>
<tr>
<td>5. Jacobian</td>
<td>6</td>
<td>6</td>
<td>29</td>
</tr>
<tr>
<td>6. Joint Acceleration</td>
<td>24</td>
<td>24</td>
<td>20</td>
</tr>
<tr>
<td>7. Inverse Dynamics</td>
<td>6</td>
<td>73</td>
<td>38</td>
</tr>
<tr>
<td>8. Inner Loop Control</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Subtotal</td>
<td>76</td>
<td>158</td>
<td>74</td>
</tr>
<tr>
<td>Equivalent V * V and V + V Operations</td>
<td>614</td>
<td>512</td>
<td></td>
</tr>
<tr>
<td>Equivalent a * b and a + b Operations</td>
<td></td>
<td></td>
<td>1932</td>
</tr>
<tr>
<td>9. LU-Decomposition</td>
<td></td>
<td></td>
<td>106</td>
</tr>
<tr>
<td>10. Solution for $\dot{\theta}_d$</td>
<td>36</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>11. Solution for $\dot{\theta}_e$</td>
<td>36</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>2110</td>
<td>1669</td>
<td>30</td>
</tr>
</tbody>
</table>

Note: All Matrices in the table are 3 x 3 matrices. All vectors are 3 x 1 vectors.
**Execution Sequencing**

The sequence of execution determines the parallel and pipeline organization. The ultimate performance of the system depends upon the amount of concurrency of the application being exploited. Concurrent execution can take various forms. The following is a list of parallelism embedded in the application.

1. Simultaneous input of both operands
2. Concurrent execution of addition and multiplication
3. Pipelined execution of successive multiplications
4. Concurrent execution of operation between vector elements
5. Concurrent execution of computation and I/O
6. Concurrent execution of subtasks within a computational module
7. Concurrent execution of computational modules

**3.2.3 Analysis of Communication Requirements**

The cost of communications within a VLSI multiprocessor system is the most important parameter to be considered during the architectural design. A communication requirement occurs when one functional element needs data from other elements. Data communications may be classified into four levels dependent upon the granularity of the functional elements between which the communications takes place.

At the lowest level, the data transfer between registers and arithmetic units takes place in almost every operation. These transfers determine the type of data bus to be used for the scalar computation unit.
At the next higher level, data transfer between scalar units is required to perform vector operations such as the vector cross product. The data transfer requirements of this level are very significant because vector computations are the predominant computations in the robotic control algorithm.

When a computational module is implemented with a group of vector units data communications between them is required. The amount of data traffic at this level depends upon the manner in which the computational module is partitioned.

At the top level, data communication is required between the computing structures for different computational modules. Data communication at the lower level is more frequent and more regular. The communication frequency and regularity decrease at higher levels. The difference in the bandwidth and complexity requirements influences the way the communication scheme is designed at each level.

3.2.4 Summary of Findings

The above discussions give a picture of the computation and communication involved in the real-time robotic control environment. The following is a summary of these findings:

1. The real-time control environment at the lowest levels is very regular and static. The tasks to be performed are usually known at the time of system configuration.

2. The control task is highly modular and lends itself to parallel implementation.

3. Single precision floating point numbers are needed to represent the control variables.
4. The predominant data structures are the $3 \times 1$ vectors and $3 \times 3$ matrices. The majority of computations are vector cross products and matrix-vector multiplications because of their computational complexities. Scalar computation is minimal, but necessary.

5. Most consecutive multiplications are independent; however, most consecutive additions are dependent.

6. Synchronized communications is suitable for data communications among processors jointly performing a well-defined task.

7. Data references are highly localized within task modules.

8. Data communications between computational modules triggers the computation activities of the receiving module.

3.3 Layered Multiprocessor Architecture for Real-Time Robotic Control Applications

There are four distinct advantages to designing the architecture with a hierarchy of layers. First, such an approach deals with design trade-offs at different levels. This enables the designer to address the design trade-off within a smaller domain where optimization is more effective and the adverse effect of the trade-off is localized. Second, the layers are built up from the lowest level. This approach offers both simplicity and flexibility. Thirdly, The layered hierarchy design allows the various elements in the hierarchy to function coherently with other parts of the system because the boundary of each layer is defined with a global perspective rather than the individual computing structure. This removes the communication bottleneck usually encountered in a multiprocessor system. Lastly, the hierarchy structure is highly modular which lends itself to effective VLSI implementation.

34
At the lowest level, this structure employs pipelining to effectively utilize the computing resources. At the next highest level, an SIMD structure is designed to exploit the parallelism between the vector elements. The SIMD layer effectively supports the vector computations required by the robotic control algorithm. The next highest level is a synchronized SIMD array structure consisting of several SIMD modules to jointly perform time-consuming tasks in the critical paths. The configuration of the SIMD module depends on the computation task it performs. At the top level is the Decentralized MIMD (DMIMD) structure consisting of several synchronized SIMD arrays. The multiple data paths in the control block diagram are mapped onto the multiple data channels of each module.

Figure 3 shows a layered multiprocessor architecture to provide the proper computing environment for the real-time robotic control application. This architecture incorporates various forms of parallel structures to achieve the maximal parallelism embedded in the applications. The following subsections present the major architectural features of each layer. Discussion of these architectural features will be general.

3.3.1 FPP Layer for Scalar Floating Point Computation

The basic element of this layer is the FPP for single precision floating point numbers. Figure 4 gives the block diagram of the FPP. The FPP includes a Floating Point Adder (FPA), a Floating Point Multiplier (FPM), and a large dual-port Register File (RF). The objective of the FPP architecture design is to achieve maximal utilization of the FPA and FPM because they are the fundamental operations of the robotic control computation.

The FPP maximizes the computational performance through overlapped execution among the FPA, FPM, and data transfer operations [5]. A dual-bus scheme
Figure 3: Layered multiprocessor architecture for real-time robotic control.
Figure 4: Layer 1 - Floating Point Processor architecture for parallel/pipelined scalar computation.
is adopted to transfer both operands in one cycle [34,35]. Both the FPA and FPM are buffered by their respective output registers.

The bus is time-multiplexed to increase the bus utilization. While the arithmetic units are performing computations, the bus can perform data transfer for other devices.

The FPM also employs a two-stage pipeline to increase the FPM computation throughput. The pipelined multiplier has the same computation bandwidth of the FPA.

The RF contains about 60 dual-port registers to increase the data reference locality. The size of the RF is chosen so that it is large enough to store most of the computation variables.

The FPP serves as the floating point computing engine of the SIMD module for vector computation. It is also incorporated with a 32-bit CPU to form the Scalar Processor Unit (SPU) for scalar computation. The detailed architectural design of the FPP will be described in Chapter 4.

3.3.2 SIMD Layer for Vector Computation

The majority of operations of the robotic control algorithm are $3 \times 1$ vector operations. The objective of this layer is to optimize the $3 \times 1$ vector computation.

The most effective parallel structure to speed up the vector operation is the SIMD structure because the operation of each vector element is identical and independent of the others. Figure 5 is the block diagram of the SIMD structure. Each SIMD node consists of three FPPs, two Shift and Broadcast Networks, two I/O Channels and a controller. During any cycle, all three FPPs execute the same operation in parallel. This structure simply triples the performance of the FPP.
Figure 5: Layer 2 - SIMD architecture for $3 \times 1$ vector operations.
Since vector operations constitute the majority of the robotic control computations, it is logical to select the SIMD layer as the basic user visible layer for the vector computations. The SIMD node can be programmed to perform all the vector computations for the robotic control system. The next chapter will describe more details of the SIMD architecture.

3.3.3 Synchronized SIMD Array (SSIMDA) Layer to Exploit Parallelism Within Task Module

The SIMD layer exploits the parallelism embedded between the vector elements. There is, however, additional parallelism at the large granularity level embedded in the robotic control algorithm. The next target for parallel implementation is the parallelism within each task module. The amount of parallelism in each module varies. Some are more sequential than others. The objective of the next layer structure is to employ several tightly coupled synchronized SIMD machines to reduce the computation time of these tasks.

Each SSIMDA machine includes a number of SIMD machines, a central Vector Sequencer (VS), a Scalar Processor Unit (SPU), four data communication Channels (CH's), and a dual-read dual-write Common Memory (CM). Figure 6 shows a synchronized SIMD Array consisting of two SIMD modules. These tightly coupled SIMD machines are connected by the two channels of each SIMD module.

The size and configuration of each synchronized SIMD array are flexible. This flexibility enables the user to tailor the configuration of the SIMD array according to the computation requirements. Chapter 7 will present methods to partition the task to achieve greater computing performance.

The operation of every SIMD module in the MIMD cluster is paced by the Vector Sequencer. The sequencer generates the next instruction address which is fed to the instruction memories of all SIMD machines. Each instruction memory
Figure 6: Layer 3 - Synchronized SIMD array architecture for the execution of tightly coupled computational tasks.
contains different instructions for the same address. The execution of these SIMD modules is thus centrally controlled. The sequencer synchronizes the operation of each SIMD module and forces them to operate in a lock-step fashion. The data communications between these modules requires no handshaking because it is already synchronized by the sequencer. This no-handshaking scheme decreases the communication delay significantly.

The SPU is a reduced-instruction-set scalar processor unit with a minimal number of instructions necessary to support basic scalar integer and floating point operations, memory and I/O operations, vector execution handling, and some control transfer operations to support a more sophisticated programming structure. The control circuit of the SPU is very simple and requires little silicon real-estate to implement.

The SPU serves two functions. First, its scalar floating point operations complement the vector computation power of the SIMD machine. Some modules, such as the LU-Decomposition for the Inverse Jacobian, requires a scalar computation capability. Second, the repertoire of the SPU instructions provides the SPU with the sophistication to act as the central controller of the synchronized SIMD array and thus greatly increases its capability.

The nature of the lock-step execution of the SSIMDA machine limits its application to the class of well-defined static tasks. As the task grows larger, so does the programming difficulty. It is more appropriate to implement task modules at this layer and leave the implementation of the entire control system to the next highest layer.
3.3.4 Decentralized MIMD (DMIMD) Layer for Complete Control Implementation

Figure 7 is the block diagram of a distributed MIMD structure with three nodes [36,37]. These nodes are connected by the four I/O channels of each SSIMD Array nodes. The size and configuration of the distributed MIMD system depend upon the target application.

The multichannel message passing communication scheme is very suitable for the proposed application. First, it increases the communication bandwidth of the processor. Second, the communication channels can be used to map the data paths in the control block diagram. This provides a natural mapping between the computing structure and the control algorithm. As discussed earlier, the regular and static nature of the real-time control environment greatly simplifies many issues involved in multiprocessor system design.

The control algorithm is partitioned into several tasks and each SSIMDA node is responsible for one or more tasks. The data path in the control block diagram is mapped by the data channels of each SSIMD node. The amount of data traffic between these nodes is very small if the tasks are properly assigned. Data passed between these nodes are in a packet form with a header identifying the destination address and packet ID. Each input data packet through the data channel causes the SPU to be interrupted and thus triggers the necessary activities in the receiving node. This form of data-driven computation requires no global controller and has the flexibility to perform a very complex control task.
Figure 7: Layer 4 - Decentralized MIMD architecture with three nodes for real-time robotic control.
3.4 VLSI Implementation - the Robotic Vector Processor (RVP) and Robotic Scalar Processor (RSP)

The previous section presented the architectural hierarchy of the layered multiprocessor system based upon the target robotic applications. Even though it was not explicitly mentioned, the implication of VLSI implementation was thoroughly evaluated in every step of architecture design.

VLSI technology has two major implications on the architectural design. First, the off-chip data communications is so much slower than the on-chip communications that its delay cost needs to be considered during the process of chip function selection. This implies that functional elements with frequent data communication between them should be incorporated into one chip. Secondly, chips with more, simpler functional elements may be better than chips with fewer but more complex functional elements. Thirdly, the number of types of the VLSI chips should be minimized to avoid extensive design cost. The function implemented by a VLSI chip should be capable of repeated use. This suggests that the system architecture should employ multiple identical structures to reduce development cost.

The block diagram in Figure 7 suggests two VLSI chips, the Robotic Vector Processor (RVP) and Robotic Scalar Processor (RSP). Both chips are used as building blocks to construct various computing structures for different applications. The "restructurability" [38] property of this architecture matches very well with the real-time control applications for two reasons. First, it provides a cost effective way to improve various computations with special computing structures. Second, the static computation requirements of the real-time control applications greatly reduce the effects of inflexibility of these special computing structures.

1. Robotic Vector Processor (RVP): The RVP implements the entire SIMD
node on a single chip. By doing so it keeps all of the data shifting for the vector operation within the chip. The elimination of off-chip traffic not only increases the data shifting speed, but also eliminates the need of a large number of I/O pins. The RVP can be used repeatedly to form various computing structures optimized for different tasks. Chapter 4 will describe the detailed architecture of the RVP.

2. Robotic Scalar Processor (RSP): The RSP chip includes the functions necessary to support the synchronized SIMD array structure. These functions include the SPU, VS, CH's, and CM. The RSP acts as the central controller of its associated RVP(s). It isolates the RVP(s) from the complexity of the control algorithm. Chapter 5 will discuss the detail of the RSP architecture.

3.5 Summary

This chapter has presented a layered hierarchical multiprocessor architecture for the real-time robotic control applications. It started with an evaluation of the computation and communication requirements of the applications, and followed with an architecture of four layers to match them. Each layer exploits different types of concurrency embedded in the application. The resulting architecture is a heterogeneous parallel structure capable of offering both the computational power for the real-time requirement and the flexibility for the wider range of applications.

The layered hierarchical multiprocessor architecture has the following characteristics:

1. It proposes a layered computing hierarchy to deal with the control algorithm at the proper level. At the Floating Point Processor level, extensive overlap is employed to maximize the use of the computing resources. At the
Single-Instruction-Multiple-Data stream processor level, it optimizes the fundamental vector operations by adapting the most effective vector computing structure. At the next layer, it exploits the embedded parallelism of well-defined computation tasks through an SSIMDA structure. At the top level, the Decentralized Multiple-Instruction-Multiple-Data stream (MIMD) architecture maps the application algorithm using the multiple communication paths and exploits the parallelism between task modules. The execution of each MIMD node is triggered by the incoming data-packet. This data-driven behavior is very similar to the behavior of the real-time control environment.

2. The computing structure is flexible and scalable. A computation intensive algorithm may be implemented with more lower level computing elements. On the other hand, a complex algorithm may be performed by more asynchronous nodes.

3. The entire structure is built upon two VLSI chips, the RSP and the RVP. These two chips are used repeatedly to configure a structure that matches the application.

The next two chapters will present a more detailed discussion on the architecture design of the two VLSI processors, the Robotic Vector Processor and the Robotic Scalar Processor.
CHAPTER IV

ARCHITECTURE OF ROBOTIC VECTOR PROCESSOR

4.1 Introduction

Chapter 3 presented the general architecture of the layered hierarchical multi­processor system for real-time robotic control. It defined four layers of computing structures with lower layers serving as the foundation of the upper ones. It also defined two VLSI chips, the Robotic Vector Processor (RVP) and the Robotic Scalar Processor (RSP), based upon the communication requirements and VLSI constraints.

The RVP implements the lower two layers. It implements the SIMD machine containing three FPPs on a single chip. This way, most of the data communications are restricted to within the chip and enjoy the high on-chip communication bandwidth. This is, however, considerable amount of circuitry on one chip. Some trade-offs must be made in order to achieve this objective.

Section 4.2 will present the block diagram of the RVP. It identifies the major functional blocks for the processor. The RVP architecture uses three identical FPPs. Section 4.3 will describe the detailed architecture of the FPP. Issues, such as the bus structure, pipelining and register file design, will be discussed thoroughly.

The function of a processor is best described by its instruction set. Section 4.4 will describe the instruction set of the RVP in detail. The objective of the RVP architecture is to pack as many functions as possible into one chip and to
exploit maximal parallelism. It is important to examine the actual parallelism offered by the processor after all the trade-offs are made. Section 4.5 will evaluate the parallelism offered by the designed RVP architecture.

Since the second layer implemented by the RVP is the foundation of the next layer, the Synchronized SIMD Array (SSIMDA) structure, the architecture design also has to consider the interface to the higher layer. The interface includes both process coordination and data communications. Section 4.6 will describe these design considerations.

4.2 RVP Block Diagram

Figure 8 is the block diagram of the RVP. The RVP architecture contains a controller, three identical Floating Point Processors (FPPs), two Shift and Broadcast Networks, and two I/O Channels.

Each FPP includes a 56 x 32 Register File, a Floating Point Adder (FPA), and a two-stage Floating Point Multiplier (FPM). Each RF stores an element of a vector. The storage capacity of the three RFs is thus fifty-six $3 \times 1$ vectors. This large amount of storage capacity is needed to increase the computation locality and to reduce the communication requirements. Section 4.3 will discuss the architectural design of the FPP in more detail.

During vector computation, each FPP operates on its respective vector element, and all three FPPs collectively perform the desired vector computation. This scheme greatly simplifies both the hardware and software design because it allows designers to conceive vector operations as scalar operations.

During vector addition and multiplication each FPP operates on data located in its own FPP. There is no need of data communications between the FPPs. However, for vector cross product, vector dot product, or matrix-vector multiplication,
Figure 8: Robotic Vector Processor block diagram.
the vector elements in one FPP are used to compute the results in other FPPs. Data communications between FPPs is, therefore, critical to the performance of the vector computation. The dual Shift and Broadcast Networks (SBN) perform the data communications between FPPs by shifting the vector elements circularly around the FPPs or broadcasting an element to all three FPPs. The broad bandwidth of the on-chip SBNs plus the overlapping of floating point operations and data communications practically eliminate the cost of data communications at the SIMD level.

I/O communications is a major consideration for vector processor design of any kind. The objective is to move the data in and out of the processor quickly without severely degrading the computation performance. The RVP includes two 32-bit wide I/O Channels, CHA and CII. Each channel consists of three 32-bit shift registers. These registers are addressable by the three FPPs as R0 or R1. The I/O channel requires 3 cycles to move a vector in or out of the RVP. However, it operates independently of the FPPs and requires only one CPU cycle to communicate to the FPPs. The choice of two I/O channels for each RVP is based upon the evaluation of communication requirement at the computational module level. Section 4.6 will discuss this issue in more detail.

4.3 FPP Architectural Design

Figure 9 is the block diagram of the FPP. The goal of the FPP architectural design is to achieve maximal utilization of the FPA and FPM. This involves the design of the FPM and FPA pipelines, the bus structure for concurrent data transfer, register file that supports the bus structure, and special registers that support the overlapped execution of the FPPs and data communications.
Figure 9: Floating Point Processor block diagram showing special addressable registers.
4.3.1 Floating Point Adder and Multiplier

The FPA and FPM are the engines of the RVP. The speed of these two functional units directly determine the speed of the RVP.

Assuming other ways to improve the circuit performance have been exploited, there are two ways to improve the FPA and FPM performance, shortening the execution time and increasing the initiation rate. The execution time is determined by the delay of the critical path in the circuitry. Typically, it is very costly to shorten the execution time because this usually requires converting serial circuits into more complex parallel circuitry. It is, however, easier to increase the initiation rate by breaking up the FPA and FPM computation into several pipeline stages. The extra hardware cost is moderate if the stages are chosen properly. Theoretically, the speed improvement could be as good as the original non-pipelined execution time divided by the execution time of the longest pipeline stage. The actual performance of the pipelined FPA and FPM depends on the application because the data dependency embedded in the application tasks may prevent the processor from taking full advantage of the pipelining. Therefore, it is important to determine the performance objective of the FPA and FPM before the actual design.

Section 3.2 has revealed that most consecutive FPA operations are data dependent. The goal of the FPA design is to minimize the execution time, and the introduction of pipeline stages in the FPA will only increase the FPA execution time. This leads to the choice of single stage pipeline for the FPA. Chapter 6 will discuss the design and layout of the FPA in VLSI.

Section 3.2 has also shown that the number of FPM operations are comparable to the number of FPA operations; and more importantly, most consecutive
FPM operations are independent. Floating point multiplication of single precision floating point numbers is slower than floating point addition because it involves a time-consuming 24 by 24 integer multiplication. However, the independence between successive FPM operations presents an opportunity to improve the FPM performance through pipelining.

The FPM design employs a two-stage pipeline scheme. The first stage performs carry-save additions of the two 24-bit mantissas. The second stage performs the carry-propagation addition on the two 23-bit numbers from the first stage. Further subdividing the first stage could result in a faster floating point multiplication. However, the area penalty caused by the introduction of the extra stages would be so great that not enough silicon area would remain to implement all of the necessary functions for the SIMD machine. Furthermore, the additional pipeline stages may not produce significant improvements on the computation throughput because the targeted computation sequence may not be able to fully realize the increased initiation rate. Chapter 6 will describe the detailed design and layout of this FPM.

The output register of the FPA, R6, is connected to both BA and BB. This allows the subsequent instructions to use the FPA result from either bus. The output registers of the FPM are arranged differently. The result of the current FPM operation is stored in R7 which is connected to BB only. The result from the previous FPM operation currently stored in register R7 will be transferred to another register $R7_d$ which is connected to BA. This output arrangement is quite useful for vector computation in which the results of two consecutive FPM operations will be used as the operands of the immediately following FPA operation. The two output registers, R7 and $R7_d$, eliminate the need to store the previous FPM result into the register file and reduce the data bus traffic drastically.
4.3.2 Dual-Bus Structure

The data bus is used to move the data among the various functional blocks connected to it. The objective of the bus configuration is to accommodate these functional units with the least delay and cost. Since the bus configuration directly relates to the design of the register file, the evaluation of the bus configuration has to consider the cost of the register file at the same time.

A single-bus structure has the advantage of being the simplest and the least costly. However, its lower bandwidth usually disqualifies it from the high performance processor design.

On the other hand, the three-bus configuration with two busses to feed the operands to the FPA or FPM and a third bus to send the resulting data back to the register file needs the support of a three-port register file. The large silicon area occupied by the three-port register file is so costly that it either limits the size of the register file or excludes other useful functions from the RVP chip.

Furthermore, in most parts of a computation, the results of the FPA and FPM are usually intermediate results, and are immediately consumed by the following operations. The third bus can be effectively replaced by registers that hold these intermediate results temporarily for the following operations.

Therefore, the FPP architecture adopts a dual-bus architecture. This bus structure exploits the parallelism between the input operands of the computation by using both busses to transfer the operands to the functional units. Output registers of the FPA and FPM are employed to reduce the traffic flowing back to the register file.

The duality of the bus structure exploits the space parallelism of the operand input. There is another dimension of parallelism not yet exploited, the temporal
parallelism. The data bus has a much larger bandwidth than the computation bandwidth of the FPA or FPM. It is natural that one would time-multiplex the data bus to increase the bus utilization, and thus maximize the utilization of the FPA and FPM.

The degree of multiplexing depends upon the data transfer requirements of the functional units connected to the busses. Since the RVP architecture is a single cycle instruction machine, the operands for a given instruction occupy the data busses for only one cycle. The data busses will be idle during the execution of instructions which require no bus data transfer. These idled cycles should also be considered for the determination of bus time-multiplexing.

Other factors such as the existence of special registers for latching of temporary results and the performance of the register file also influence the choice of cycle time. Assuming the FPM has the same computation bandwidth as the FPA, the data bus cycle time is chosen to be one-fourth of the FPA execution time. This way, the bus uses half of its cycles to supply operands to the FPA and FPM, and uses the other half for data transfer among the FPA, FPM, RF, SBNs, and CHs. This design choice is subject to the justification of the register file design and the applications.

4.3.3 Dual-Port Register File

As was discussed in the previous subsection, the choice of bus configuration depends on the cost of implementation of the associated register file. To support the dual-bus architecture, a dual-port register file which satisfies both the bandwidth and size requirements must be designed.

The size of the RF influences the data locality and hence the I/O requirements. Ideally, the RF should be as large as possible so that all data reference other than
input and output is local. However, as the size of the RF increases, so does the RF access delay. The size of the register file must be weighed against the RF access delay and other functions needed to be included in the chip.

The RF of the FPP contains 56 dual-port registers which are sufficient to hold most frequently accessed data. As will be discussed later, there are eight special registers addressable as R0 through R8. Thus the total number of addressable registers in the RVP is 64.

The register file has to have two read ports to support concurrent input of operands. Duality is not required for register write because of the data reduction after the computation. This allows compact register cell design by modifying the common 6-T memory cell. Chapter 6 will present the register file design in detail.

4.3.4 Special Addressable Registers

Each FPP has eight special registers associated with the functional units. These special registers are addressable as R0 through R8. They are the two I/O Channel registers, two SBN registers, two SBN broadcast registers, and the output registers of the FPA and FPM.

The two I/O channel registers, R0 and R1, are shift registers for the I/O channels. In one cycle, data can be transferred between the FPPs and the I/O channels. These registers allow the I/O operation to proceed independently of the FPP operations.

The SBN shift registers, R2 and R3, perform the data transfer between the FPPs. In addition to the shift registers, there are two broadcast registers, R4 and R5, which are useful for the executions of matrix-vector multiplications. A broadcast register is implemented by three pass transistors which connect one of the three shift busses to the three FPP data busses (BA or BB). Whenever
R4 or R5 is addressed as a source register, the processor controller turns on the corresponding pass transistors and shift the data forward in the SBN at the same time. Figure 10 is a more detailed data path block diagram of the RVP showing the SBN shift and broadcast registers. The existence of the shift and broadcast registers allows the RVP to perform vector computation with little communication delay.

The FPA and FPM output registers, R6 and R7, store the temporary results of the FPP operations and thus eliminate the write-RF operation at the end of each instruction. They not only shorten the instruction cycle by eliminating the write-RF operation, but also reduce the bus traffic. The existence of these registers effectively eliminates the need of a third bus without any significant performance degradation.

The existence of registers R0, R1, R6, and R7 also simplify the RVP control logic because they buffer the results of multi-cycle operations, such as the I/O and floating point operations. This way, only the first execution cycle of every instruction uses the data bus, and the controller does not have to keep track of the state of the previous instructions. This "fire-and-forget" control scheme makes the processor look like a single instruction cycle machine. The reduction of control complexity is critical in the RVP design because it saves valuable silicon area for the implementation of FPPs.

These registers, however, have two constraints that need to be fully understood in order to use them properly. The first constraint is that registers R0, R2, R4, and \( R7_d \), are associated with Bus BA and registers R1, R3, R5, R7, are associated with Bus BB. Direct data transfer between registers on different busses is not permissible. The other constraint is that half of the registers, R4 though R8, are
Figure 10: RVP data-path block diagram showing special addressable registers.
not writeable because they are the output registers of the FPA, FPM, and Shift and Broadcast Network.

4.4 RVP Instruction Set

The RVP is an SIMD machine which allows all three FPPs to execute the same instruction at the same time. Each instruction appears as a single cycle instruction to the processor control because each functional block is responsible for its own sequencing. It is the responsibility of the software designer to keep track of the state of the computation process.

The architecture of the RVP is register-oriented, because it yields the fastest program execution. All instructions are 16-bit wide which permits a simple and efficient instruction fetch-and-execute mechanism. The instruction format is simple, with fields at fixed locations, for simple and fast instruction decoding.

Figure 11 shows the instruction set of the RVP. The binary and symbolic opcodes of the 7 RVP instructions are shown in Figure 11(a). The opcode uniquely determines the interpretation of the instruction format. Figures 11(b) and 11(c) describe the instruction formats and operands.

The instruction set can be divided into two categories: dual-port instructions and single-port instructions. The dual-port instructions take advantage of the dual-port Register File, the dual Shift and Broadcast Network, and the dual I/O Channels. During the execution of dual-port instructions, the operands propagate on two busses. The sources and destinations of these dual data paths are subject to the constraints of the processor structure. The majority of the instructions used in the applications belong to this category.

There is one single-port instruction, Write register File (WF), to move the data from one of the eight special registers to the register file. This is a single-port
a) Instruction opcodes

<table>
<thead>
<tr>
<th>Opcodes</th>
<th>00 XX</th>
<th>01 XX</th>
<th>11 XX</th>
<th>10 XX</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>NOP</td>
<td>WF</td>
<td>FPA</td>
<td>OUT</td>
</tr>
<tr>
<td>01</td>
<td>NOP</td>
<td>NOP</td>
<td>FPS</td>
<td>NOP</td>
</tr>
<tr>
<td>11</td>
<td>SH</td>
<td>NOP</td>
<td>FPM</td>
<td>NOP</td>
</tr>
<tr>
<td>10</td>
<td>NOP</td>
<td>NOP</td>
<td>LSR</td>
<td>NOP</td>
</tr>
</tbody>
</table>

b) Instruction format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>FPA, FPS, FPM, LSR</td>
<td>1 1 x x</td>
</tr>
<tr>
<td>WF</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>SH</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operands</th>
<th>Rs</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Rs</td>
<td>Rs</td>
</tr>
<tr>
<td>WF</td>
<td>0-7</td>
<td>8-63</td>
</tr>
<tr>
<td>FPA/FPS</td>
<td>8-63,0,2,4,6,7d</td>
<td>8-63,1,3,5,6,7</td>
</tr>
<tr>
<td>FPM</td>
<td>8-63,0,2,4,6,7d</td>
<td>8-63,1,3,5,6,7</td>
</tr>
<tr>
<td>OUT</td>
<td>8-63,0,2,4,6,7d</td>
<td>8-63,1,3,5,6,7</td>
</tr>
<tr>
<td>LSR</td>
<td>8-63,0,2,4,6,7d</td>
<td>8-63,1,3,5,6,7</td>
</tr>
</tbody>
</table>

Msh/Mshb (Mode -- Shift)

01 : Forward shift
11 : Backward shift
x0 : No shift

Figure 11: Robotic Vector Processor instruction set.
operation because the Register File allows only one write at any cycle.

Each of the seven instructions will be discussed in the following paragraphs:

1. **WF** (Write register File) writes the content of one of the eight special addressable devices, R0 through R7, to the Register File. When the source registers are R0 and R1, the **WF** instruction stores the input data to the RF. When the sources are R6 and R7, the **WF** instruction stores the results of the **FPA** and **FPM** operations. Since data in the Register File can be accessed by both busses, this instruction is the only instruction to bridge the data between the two busses.

2. **OUT** (OUTput data) outputs the data through the dual I/O channels. Data are moved from Rsa to Channel A through bus BA, and moved from Rsb to Channel B through bus BB. When Rsa (or Rsb) is less than two it indicates no output for Channel A (or Channel B). The output instruction takes four cycles, but the last three cycles are executed independently by the channel controller.

3. **FPA** (Floating Point Add) reads the operands from both busses and executes the floating point add operation. Both operands specify the source registers. The data are latched by the input registers of the Floating Point Adder. The result of the **FPA** operation is stored in the special register R6.

4. **FPS** (Floating Point Subtract) is similar to the **FPA** instruction except that the subtract function will be performed.

5. **FPM** (Floating Point Multiply) reads the operands from both busses and executes the floating point multiplication operation. The result of the **FPM**
operation is stored in the special register R7 and the previous result is moved to register R7q.

6. *LSR* (Load Shift Registers) loads the dual shift registers through both busses. The source registers are subject to the same constraints as the *FPA*, *FPS*, and *FPM* instructions. This instruction is useful for the computations of vector cross product and matrix-vector multiplication.

7. *SH* (Shift) shifts the data in the dual SBNs. Both SBNs are controlled independently. The three shifting modes are forward circular, backward circular, and no shift. During vector cross product computations, the two shifters shift in the opposite directions.

### 4.5 RVP Parallelism

The previous sections have presented the architecture of the RVP. The considerations of parallel execution, both parallel and pipelining, are evident in every aspect of the processor design. This section will summarize the parallelism realized by the RVP architecture.

#### 4.5.1 The Various Forms of Parallelism of the RVP Architecture

This subsection presents the various forms of parallelism exploited by the RVP architecture. Figure 12 shows the pipelining scheme of the RVP. Each instruction begins with an instruction fetch cycle, followed by an operands fetch cycle (or data transfer cycle), and is ended after zero to five execution cycles.

Since the results of all the multiple cycle instructions are buffered, each instruction only needs to use the data busses once, or not at all. The processor fetches a new instruction at every cycle, including the *NOP* instruction. This results in an architecture of heavily overlapped execution.
Typically, pipelining in a data path module is employed only to the degree that it helps to alleviate a severe bottleneck in system performance. The concurrency provided by the RVP comes mainly from the overlapped execution among various data path modules. In fact, FPM is the only pipelined data path module with two stages. This is due to the data dependencies between successive operations as well as the area penalty of pipelined implementation.

**Overlapped Instruction Fetch**

The instruction fetching overlaps execution of the previous instruction. This overlapping is very cost effective because of the little extra hardware necessary to implement it.

**Dual-Bus Parallelism**

The dual-bus configuration of the FPP offers parallelism between the two busses. Either bus has its associated Register File port (for read), Shift and Broadcast Network, and I/O Channel. The two data busses reduce the time for operand fetches to only one cycle. Six of the seven instructions in the instruction set take advantage of the dual-bus parallelism.

**FPM Pipeline**

The first instruction in Figure 12 is the FPM instruction. After the instruction fetch and register read cycles, the FPM uses the next four cycles for carry-save-adder multiplication, which are the first stage of the FPM operation. The second stage of the FPM is the carry propagation addition (CPA). The two-stage pipeline increases the FPM computation rate by taking advantage of the independence of successive FPM operations.
Overlapped FPA Execution with FPM execution

The time-multiplexing of the data bus allows for overlapped execution of the FPA with the FPM operations. The execution of FPA requires three cycles. This overlapped execution scheme often eliminates the computation delays of the FPA operations.

Overlapped Data Transfer

The time-multiplexing of the data bus allows the overlapped execution of the data transfer, such as the WF and LSR operations. This overlapped execution scheme almost eliminates the cost of the data transfer within the RVP.

Parallel Execution of Three FPPs

All instructions, except SH and IO, take place in each of the three FPPs. This array processing fully exploits the spatial parallelism of the vector computation. This SIMD structure improves the performance of $3 \times 1$ vector computation by a factor of three with the cost of three FPPs and one controller. The saving of the two controllers makes the speed-area ratio of the RVP greater than 1.

Overlapped FPP and IO Operations

Each I/O channel has its own controller to handle the execution of data input and output. Therefore, it may overlap with the FPP operations. Although the IO instruction requires four cycles of execution time, it only consumes one RVP instruction cycle. These overlapped FPP and I/O executions fully utilize the bandwidth of the data busses and I/O channels. Further discussion on overlapped execution is included in the next subsection.
These numbers indicate the degrees of parallelism being exploited by these instructions in Cycle 6.

Figure 12: Robotic Vector Processor Pipeline.
4.5.2 Maximal Parallelism of the RVP

With all the parallelism embedded in the RVP, the most important performance measurement is the resulting system throughput. This subsection evaluates the peak performance of the RVP.

If the RVP instruction sequence is as shown in Figure 12, the RVP exhibits its maximal parallelism at cycle 6. The parallelism includes:

1. Both FPM pipeline stages are executing,
2. FPA is executing,
3. Both busses, BA and BB, are transferring data,
4. All three FPPs are executing in parallel,
5. Both I/O channels are transmitting data, and
6. RVP is fetching the next instruction.

The degrees of parallelism of items 1 through 3 are multiplied by 3 because of the concurrent execution of three FPPs. The total degrees of parallelism exhibited during cycle 6 is eighteen. When FPM and FPA are fully utilized the RVP performs three FPM and three FPA operations every four cycles. For a targeted cycle time of 125ns, this represents a computing power of 12 MFLOPs.

The effective RVP throughput depends heavily upon the application. Chapter 7 will present the implementation of the robotic control scheme using one or several RVPs.
4.6 RVP Control and Data Communications for the Formation of Synchronized SIMD Structures

Although the RVP offers enormous parallelism for the computation of robotic control algorithms, it must still work with other RVPs to meet the real-time performance requirement. This is the Synchronized SIMD layer described in the previous chapter.

Data communications and process synchronization are the two key issues for the formation of the processor array. The off-chip data communications between VLSI processors is slower and has an adverse impact on the overall performance. The process synchronization between RVPs determines how these processors cooperate to share the computation responsibility. These issues have to be carefully evaluated during the RVP architectural design.

To deal with the data communication requirement, the RVP includes two 32-bit I/O channels. The two channels not only increase the data communications bandwidth but also increase the flexibility of the SIMD structure. They allow the RVP array to be configured according to the data requirements of the application. Chapter 7 will present examples on how to use several RVPs to improve the performance of vector computations.

A processor controller usually contains two parts, the next instruction address sequencer, and the instruction decoder. The sequencer controls the sequencing of the execution. To achieve tight synchronization among RVPs, the RVP architecture separates the sequencer and the decoder circuit. The sequencer is removed from the RVP and is placed in the RSP chip. All RVPs belonging to the same SIMD array are controlled by the same sequencer. This creates a very tightly synchronized computing structure. It also saves a considerable silicon area for the RVP.
4.7 Summary

This chapter presented the architectural design of the Robotic Vector Processor (RVP). Various forms of low level parallelisms were exploited during the design of the processor. The architecture carefully handles these parallelisms so that they do not interfere with each other. This results in a peak performance of eighteen degrees of parallelism.

The architecture consideration for the construction of a synchronized RVP array was also presented. The next chapter will describe the architecture of the higher layers in more details.
CHAPTER V
ARCHITECTURE OF ROBOTIC SCALAR PROCESSOR

5.1 Introduction

Chapter 3 has presented the architectural features of a four-layer multiprocessor system built upon Robotic Vector Processors and Robotic Scalar Processors. Chapter 4 has described the architectural design of the RVP, the computing engine of the restucturable architecture. This chapter presents the architectural design of the RSP which provides the control, communications, and scalar computations to support the computational power of the RVP. The RSP architecture includes four major components: Common Memory (CM), Scalar Processor Unit (SPU), Vector Sequencer (VS), and data communication Channels (CH's).

Each RSP, together with several associated RVPs, forms a Synchronized SIMD Array (SSIMDA) to perform synchronized computation for a control system or a portion of it. There are two key issues, the data communications and the control coordination, that need to be addressed in order to maximize the parallel executions among the vector, scalar, and I/O operations. Section 5.2 presents a CM structure which provides the data communications among these components.

Although the CM provides the vehicle for data communications among the various components, the SPU is the component that supervises these interface activities. Section 5.3 describes the architecture of the SPU. The SPU adopts the UC Berkeley RISC [8] architecture as its basic architecture and provides the
basic instructions necessary to perform complex computations. Added to the basic architecture are the floating point processor and circuitry to provide the interfaces between the SPU and the other components of the RSP, namely, the CM, VS, and CH's.

Section 5.4 describes the architecture of the VS which is the central sequencer for the control of RVP operations in the SSIMDA. The I/O communications scheme between RSPs is presented in Section 5.5. Both the VS and I/O channels operate independently, however, under the supervision of the SPU to ensure proper coordination among them. Section 5.6 presents the parallel execution of the scalar computations, vector computations, and I/O communications.

5.2 Common Memory

Figure 13 is the block diagram of the RSP. The data communications between the SPU, RVPs, and other RSPs are implemented by the CM structure. On the other hand, the control of these communications is supervised by the SPU. This section describes the architectural features of the CM.

The CM is a full dual-port memory with two busses, CMBA and CMBB. Each bus has full read and write access to the CM. RVPs are connected to bus CMBA through CHV. The SPU and four I/O communication channels are connected to bus CMBB. All data transfers take place between the CM and these devices. This scheme greatly simplifies the communications between the SPU, RVPs and other RSPs by converting complex interfaces between devices to simple memory accesses.

Figure 14 is the timing of the CM busses. Bus CMBA is dedicated to the RVPs because it serves an array of RVPs. Bus CMBB is time-multiplexed to serve the SPU and four I/O channels. The SPU uses half of the bus bandwidth and the four I/O channels share the remaining half. Each device accesses CMBB at its assigned
Figure 13: Robotic Scalar Processor block diagram.
Figure 14: Common Memory bus time-multiplexing scheme.
time slots. The CM bus bandwidth assignments are determined according to the data transfer requirements of RVPs, SPU, and I/O communications. By dedicating bus CMBA to RVPs, the CM provides the RVPs with a fully accessible memory. Also, the CM gives full access to the SPU by assigning half of the bandwidth of CMBB to it because the SPU operates at half of the RVP clock rate. Each I/O channel has a 4-bit external bus which matches the allocated CM bus bandwidth.

The fixed CM time-multiplexing access scheme has two advantages:

1. It is simple. Complicated bus arbitration circuitry is avoid, valuable silicon real-estate is allocated to other critical functions.

2. It is transparent. Each interfaced device considers the CM as its own memory to read from and write to.

3. It is deterministic. The delays of the data communications are deterministic which is a very desirable feature for real-time control systems.

The size of the CM depends upon the application. For the Inverse Plant Plus Jacobian robotic control algorithm, the variables communicated between modules as described in [3] are 187. In addition to these variables, there are many parameters used during the computation process that need to be stored in the CM. The design size of the CM is 512 words which is sufficient to meet the computation and I/O requirements.

5.3 Scalar Processor Unit

The standard functions of the SPU include the ALU operations, control transfer, memory interface, and miscellaneous operations. In addition to these standard
features, the RSP must handle floating point computation, CM interface, Vector Sequencer interface, and global communications.

To implement all these functions on only a portion of the silicon chip, the most logical decision is to include only the most essential features for the SPU. The best approach for this is to choose the UC Berkeley RISC architecture [8] as the SPU basic architecture. The SPU eliminates some of the non-essential features from the RISC architecture and adds the features for floating point computation, VS interface, CM interface, and I/O communications.

Since the SPU adopts the RISC architecture as the basic architecture, this section will briefly describe the basic features of the SPU and concentrate on the special features for the RSP. Details of the RISC architecture design may be found in [8].

5.3.1 The SPU Data Path

Figure 15 is the data path block diagram of the SPU. This is a dual-bus processor with a large register file, ALU, a floating point adder, and a floating point multiplier. Its basic components are the following:

- Register File: 80-word by 32-bit register file, with a dual-port address decoder and with latches Ra, Rb, Rd holding the register addresses from the instruction.

- Four Special registers: R0 is hardwired to contain zero. R1 is the Program Counter (PC). R2 is the output registers of the FPA. R3 is the output register of the FPM. (R3d is the delayed version)
Figure 15: Scalar Processor Unit block diagram.
- **PSW0**: a 14-bit Processor Status Word. It includes Current Window Pointer (CWP), Saved Window Pointer (SWP), Interrupt enable, Overflow, Negative, Zero, Carry, VS Jump (VSJ), and Vector Status Code (VSC).

- **PSW1**: This is the 24-bit Processor Status Word for I/O channels. A 6-bit field is assigned to each channel which includes a 4-bit input packet ID, a ready-to-send bit, and a ready-to-receive bit.

- **DST**: the Destination latch, serving as the temporary pipeline latch. The result of each cycle's operation is held there, until that result is written into the register file, or otherwise used, during the next cycle.

- **AI, BI**: the two input latches of the ALU.

- **ALU**: a 32-bit integer Arithmetic and Logic Unit.

- **NEXTPC**: the Next-Program-Counter register, which holds the address of the instruction currently being fetched.

- **INC**: an incrementer

- **PC**: the Program-Counter register, which holds the address of the instruction currently being executed. The PC is addressable as R1.

- **LSTPC**: the Last-Program-Counter register, which holds the address of the instruction last executed.

- **IMM**: The IMMEDIATE latch, which holds the 13 LS-bits of the incoming instruction.

- **DIMM**: Data/IMMediate combined latch, which holds the data from the memory, or immediate operands being forwarded to the data-path.
• **OP**: the 6-bit OPeration-code of the instruction.

• **Ba, Bb**: the register file busses.

• **BA, BB**: the busses to feed AI and BI, etc.

• **BO**: the bus to route address and data to the pads.

• **BE**: the off-chip bi-directional time multiplexed address/data bus, which connects the SPU to the memory.

• **FPA**: the Floating Point Adder, which performs 32-bit single precision floating point addition and subtraction. The output latch of the FPA is addressable as R2.

• **FPM**: the Floating Point Multiplier, which performs 32-bit single precision floating point multiplication. The output and the delayed output latches of the FPM are addressable as R3 and R3d.

• **CMINT**: the Common Memory INTerface buffer.

### 5.3.2 Instruction Set

The architecture of the SPU is register-oriented. All instructions have a fixed width of 32 bits for simplicity and efficiency of the instruction fetch-and-execute mechanism. The instruction format is simple and fixed for fast instruction decoding. Since the RSP must support floating point operations, vector sequencing, and I/O communications, the instruction set also includes instructions for these functions.

Figure 16 is the instruction set of the SPU. There are 26 instructions in the instruction set. Only the essential and frequently used instructions are implemented to reduce the control complexity and execution time.
<table>
<thead>
<tr>
<th>Grp</th>
<th>Instructions</th>
<th>Operands</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>sll</td>
<td>Rd,Ra</td>
<td>Rd ← Ra shifted by 1 shift left</td>
</tr>
<tr>
<td></td>
<td>src</td>
<td>Rd,Ra</td>
<td>Rd ← Ra shifted by 1 shift right arithmetic</td>
</tr>
<tr>
<td></td>
<td>srl</td>
<td>Rd,Ra</td>
<td>Rd ← Ra shifted by 1 shift right logical</td>
</tr>
<tr>
<td></td>
<td>and</td>
<td>Rd,Ra,Sb</td>
<td>Rd ← Ra and Sb bitwise logical AND</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>Rd,Ra,Sb</td>
<td>Rd ← Ra or Sb bitwise logical OR</td>
</tr>
<tr>
<td></td>
<td>xor</td>
<td>Rd,Ra,Sb</td>
<td>Rd ← Ra xor Sb bitwise logical XOR</td>
</tr>
<tr>
<td></td>
<td>add</td>
<td>Rd,Ra,Sb</td>
<td>Rd ← Ra + Sb integer add</td>
</tr>
<tr>
<td></td>
<td>addr</td>
<td>Rd,Ra,Sb</td>
<td>Rd ← Ra + Sb + Carry integer add with carry</td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td>Rd,Ra,Sb</td>
<td>Rd ← Ra - Sb integer subtract</td>
</tr>
<tr>
<td></td>
<td>subc</td>
<td>Rd,Ra,Sb</td>
<td>Rd ← Ra - Sb - Borrow integer subtract with borrow</td>
</tr>
<tr>
<td>2</td>
<td>call</td>
<td>(Rx)Sb</td>
<td>Rd ← PC; PC ← Rx+Sb,CWP← call,indexed, change window</td>
</tr>
<tr>
<td></td>
<td>jump</td>
<td>Cond, (Rx)Sb</td>
<td>if Cond then PC ← Rx + Sb cond. jump,indexed, delayed</td>
</tr>
<tr>
<td></td>
<td>return</td>
<td>(Rx)Sb</td>
<td>PC ← Rx + Sb; CWP++ return, change window</td>
</tr>
<tr>
<td>3</td>
<td>ld</td>
<td>Rd, (Rx)Sb</td>
<td>Rd ← M[Rx + Sb] load register</td>
</tr>
<tr>
<td></td>
<td>st</td>
<td>Rd, (Rx)Sb</td>
<td>M[Rx + Sb] ← Rd store register</td>
</tr>
<tr>
<td></td>
<td>getpc</td>
<td>Rd</td>
<td>Rd ← lastPC get last PC after interrupts</td>
</tr>
<tr>
<td></td>
<td>getswi</td>
<td>Rd</td>
<td>Rd ← PSWi read status wordi</td>
</tr>
<tr>
<td></td>
<td>putswi</td>
<td>Rd</td>
<td>PSWi ← Rd set status wordi</td>
</tr>
<tr>
<td>4</td>
<td>fpadd</td>
<td>Rd,Ra,Rb</td>
<td>Rd ← R2 ; R2 ← Ra + Rb floating point add</td>
</tr>
<tr>
<td></td>
<td>fps</td>
<td>Rd,Ra,Rb</td>
<td>Rd ← R2 ; R2 ← Ra - Rb floating point subtract</td>
</tr>
<tr>
<td></td>
<td>fpm</td>
<td>Rd,Ra,Rb</td>
<td>Rd ← R3 ; R2x ← Ra * Rb floating point multiply</td>
</tr>
<tr>
<td>5</td>
<td>rdCM</td>
<td>Rd,Acm</td>
<td>Rd ← M[Acm] read Common Memory</td>
</tr>
<tr>
<td></td>
<td>wrCM</td>
<td>Rd,Acm</td>
<td>M[Acm] ← Rd write Common Memory</td>
</tr>
<tr>
<td>6</td>
<td>forkV</td>
<td>Rd, Vaddr</td>
<td>VPC ← Vaddr set VPC to Vaddr</td>
</tr>
<tr>
<td>7</td>
<td>in</td>
<td>Ch</td>
<td>set input Ch ready-to-receive</td>
</tr>
<tr>
<td>8</td>
<td>out</td>
<td>Ch,Addr,ID,wds</td>
<td>output data packet via Ch</td>
</tr>
</tbody>
</table>

Ra,Rb,Rd,Rx: a register; Sb: Rb or a 13-bit immediate constant; Acm: an address of CM; Cond: 4-bit condition code; PC: Program Counter; CWP: Current Window Pointer; The first group of instructions(ALU) set the condition code.

Figure 16: SPU instruction set.
The instruction format is shown in Figure 17. The majority of instructions have the same instruction format. Each new instruction is fetched into the Scalar Instruction Register (SIR). SIR<31:26> is the operation code. Sixty-four registers are visible to the compiler at any one time, and thus requires 6-bit fields to specify the sources and destination.

Field SIR<25:20> may specify one of two things according to the opcode. For conditional control-transfer instructions, it specifies the branch-condition. For most other instructions, this field specifies the Rd register number.

The Sb field is 14 bits long. Its leading bit specifies whether it should be interpreted as Rb or as an immediate constant. In the former case, SIR<5:0> specifies Rb. In the latter case, a 13-bit 2’s-complement immediate constant in assumed.

The 26 instructions can be grouped into five categories: basic instruction set, floating point instructions, vector sequencer instruction, Common Memory instructions, and I/O instructions.

1. Basic Instruction Set: The basic instruction set is a stripped down version of the RISC instruction set. Removed from the original instruction set are the non-essential instructions such as the half-word and byte instructions, long immediate instructions, etc.. The basic instruction set provides only the necessary instructions to support complex softwares. These basic instructions can be divided into four groups: ALU operation instructions (group 1), control transfer instructions (group 2), memory access instructions (group 3), and miscellaneous instructions (group 4). All basic instructions except the memory access instructions are single cycle instructions.

2. Floating Point Instructions: The SPU includes the same FPA and FPM
a) Instruction Formats

<table>
<thead>
<tr>
<th>Groups</th>
<th>Instructions</th>
<th>Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ALU</strong></td>
<td>all, era, srl,</td>
<td><img src="image" alt="" /></td>
</tr>
<tr>
<td></td>
<td>add(c), sub(c)</td>
<td></td>
</tr>
<tr>
<td><strong>Control</strong></td>
<td>call, jump, reta</td>
<td><img src="image" alt="" /></td>
</tr>
<tr>
<td><strong>Transfer</strong></td>
<td>id, st</td>
<td></td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>get</td>
<td>pc, get</td>
</tr>
<tr>
<td><strong>Misc.</strong></td>
<td>fpa, fps, fpm</td>
<td></td>
</tr>
<tr>
<td><strong>Floating</strong></td>
<td>Point</td>
<td><img src="image" alt="" /></td>
</tr>
<tr>
<td><strong>CM</strong></td>
<td>rdCM, wrCM</td>
<td></td>
</tr>
<tr>
<td><strong>Vector</strong></td>
<td>forkV</td>
<td></td>
</tr>
<tr>
<td><strong>I/O</strong></td>
<td>in, out</td>
<td></td>
</tr>
</tbody>
</table>

a) Processor Status Words

<table>
<thead>
<tr>
<th>psw0</th>
<th></th>
<th><img src="image" alt="" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>psw1</td>
<td></td>
<td><img src="image" alt="" /></td>
</tr>
</tbody>
</table>

Figure 17: Scalar Processor Unit instruction formats and Processor Status Words.
designed for the RVP. Therefore, the \texttt{fpa}, \texttt{fps}, and \texttt{fpm} instructions are included in the instruction set. The floating point reciprocal may be computed by the Newton-Raphson iteration method [39]. Appendix E will describe computation of the floating point reciprocal using the ALU, FPM, and FPA functions.

3. Vector Procedure Call Instruction: The SPU can initiate a RVP procedure by executing the \texttt{forkV} instruction. This instruction forces the RVP sequencer to fetch its next instruction from the address specified by SIR<12:0>.

4. Common Memory Access Instructions: The \texttt{rdCM} and \texttt{wrCM} instructions (both are single cycle instructions) transfer data between the SPU register file and the CM.

5. I/O Communications Instructions: The I/O communications between RSPs and peripheral devices are handled by the four I/O channels. The \texttt{out} instruction causes the I/O channel to send a data packet by specifying the output channel, starting CM address, number of words to be transferred, and the packet ID. The I/O channel handles the actual data transfer between RSPs. This relieves the SPU from the time-consuming I/O operations. Section 5.5 will describe the communications scheme in more detail.

5.3.3 Multi-Window Register File

The SPU adopts a register-oriented architecture because studies have indicated that supporting fast operand accesses is the key to good processor performance. The objective of the register file organization is to maximize the usage of on-chip registers while minimizing the off-chip memory access.
The memory access decreases as the number of addressable registers increases. However, all the values in these registers have to be saved during procedure calls. Because of the frequent procedure calls in complex software programs, the organization of the register file has to provide the programmer with a large number of visible registers while minimizing the cost of the procedure calls.

The best way to find the optimal register file organization is to examine the structure of the applications programs and to allocate the registers according to the data requirements of these procedures. The data requirements can be classified into three categories: global variables, local variables, and arguments passed between procedures. The multiple, overlapping, fixed-size windows register file organization is a direct result of mapping the program data requirements onto the register file.

The Register File contains 80 dual-port registers, plus four special registers, the zero register, the PC and the output registers of the FPA and FPM. These 84 registers are arranged into four fixed-size, overlapping windows. Not all 84 registers are visible to the programmer at any one time. The ones that are visible are called 'the current window'. The Current Window Pointer (CWP) selects the current window. The register-number inputs to the register decoder are supplied by the instruction, and they selects one of the registers within the current window. Upon a procedure call, the CWP points to the next window. The CWP points back to the previous window after a return instruction.

Some registers belonging to all windows are called 'global registers'. These registers are used to store global data. Some registers belong to only one window. These 'local registers' are used to store local variables of procedures. Some registers belong to two adjacent windows. These "overlap registers" are used to pass arguments between procedures.
The computation of the robotic control applications involves a number of vectors, matrices, and scalars. Each one of the 11 program modules in [3] requires for a large number of registers because of the vector and matrix operands. This clearly indicates the need for large programmer visible window. A 64 register window size is thus chosen.

During procedure calls, the arguments passed to the child procedure are mostly pointers of the vectors and some scalar variables. It is also found that other than vector operands, the local variables are mostly counters and indexes. Most of the procedures written by Lilly [3] contain four arguments or fewer. These procedures also use fewer than four local variables most of the time. This leads to the choice of four overlap registers as well as four local registers.

Figure 18 is the organization of the multi-window Register File. Each procedure can address 64 registers in the Register File. R0 through R51 are global variable registers. Each procedure has 4 registers for local variables, 4 registers for passing arguments to its child procedure, and 4 registers for arguments passed from its parent procedure. This is the optimal register organization because it matches well with the data requirements of the application program. Further details on the multi-window scheme may be found in [8].

5.3.4 SPU Pipeline

All instructions in the SPU instruction set, except ld and st, are single cycle instructions. The floating point operations appear to the controller as single cycle instructions even though they actually take longer to execute. The objective of the pipelining is to allow the SPU to initiate one instruction each cycle. Figure 19(a) shows the pipelined execution of SPU instructions. Basically, the SPU instruction pipeline includes three stages, instruction fetch, execution, and storing of result.
Figure 18: Scalar Processor Unit register file overlapped multiwindow scheme.
During load and store instructions, the pipeline is suspended for one cycle and continues afterwards.

Most instructions are executed adhering to the following pattern:

- read Ra and Rb (or get immediate),

- perform ALU or floating point operation, and

- write the result into Rd, or use it as an effective-address for load and store instructions. These two require additional cycle to complete the memory access. For floating point operations, the result written into Rd is the result of the previous floating operation stored in the output latch.

Figure 19(b) shows the resulting data-path timing requirements. At each cycle, the register file performs a read and a write operation. The instruction cycle is thus twice as long as the RVP instruction cycle. This longer instruction cycle does not allow as much pipelined and overlapped execution as does the RVP. There are two reasons for the longer instruction cycle. First, the vector computation on the RVP requires more frequent data transfer along the busses than the SPU. Second, the SPU has branching and memory access instructions which complicate the implementation of the pipeline.

5.4 Vector Sequencer

The VS generates the next execution address for RVPs. Figure 20 is the instruction set of the VS. There are six instructions in the instruction set: Loop, Nop, Jump, SignalS, RdCM, and WrCM. Each instruction is 16 bits wide.

When the SPU executes a forkV instruction, it loads the next vector instruction address register with SIR< 13 : 0 >. The fetched instruction is held in the
a) SPU pipeline

![SPU pipeline diagram]

b) SPU data path requirements

![Data path requirements diagram]

Figure 19: Scalar Processor Unit pipeline and data path requirements.
a) Instruction opcodes

<table>
<thead>
<tr>
<th></th>
<th>00X</th>
<th>01X</th>
<th>10X</th>
<th>11X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RdCM</td>
<td>SignalS</td>
<td>Nop</td>
<td>Loop</td>
</tr>
<tr>
<td>1</td>
<td>WrCM</td>
<td>Jump</td>
<td>Nop</td>
<td>Loop</td>
</tr>
</tbody>
</table>

b) Instruction format

|        | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | . . . | . . | . . | . . | . . | . . | . . | . . | . . | . . | . . | . . | . . | 0  |
|--------|----|----|----|----|----|----|----|----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| RdCM,WrCM | 0 0 | X  |    |    |    |    |    |    |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      | 0   |
| SignalS | 0 | 1 | 0 |    |    |    |    |    |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      | VSC |
| Jump    |    |    |    | 0 | 1 | 1 |    |    |      |      |      |      |      |      |      |      |      |      |      |      |      |      | displacement |
| Loop    |    |    |    | 1 | 1 |    | X |    |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |
| Nop     |    |    |    | 1 | 0 |    | X |    |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |

Figure 20: Vector Sequencer instruction set.
Vector Instruction Register (VIR). The following describes the execution of each instruction.

- **Loop**
  
  \[ \text{PC} = \text{PC} \]

- **Nop**
  
  \[ \text{PC} = \text{PC} + 1 \]

- **Jump**
  
  if VSJ bit in PSWO is true, \[ \text{PC} = \text{VIR}<12:0> \]
  else \[ \text{PC} = \text{PC} + 1 \]

- **SignalS VSC**
  
  set VSC bits in PSWO and interrupt SPU
  \[ \text{PC} = \text{PC} + 1 \]

- **RdCM CMAaddr**
  
  \[ \text{VIR}<8:0> \to \text{CHV} \]
  \[ \text{PC} = \text{PC} + 1 \]

- **WrCM CMAaddr**
  
  \[ \text{CHV} \to \text{VIR}<8:0> \]
  \[ \text{PC} = \text{PC} + 1 \]

The **Jump** instruction performs a conditional jump based upon the VS Jump bit in PSWO. The SPU sets or clears this bit through the *putpsw0* instruction. This is useful to schedule the vector computation. For example, the SPU may initiate the vector computation when the first set of data is available. When the data for the later part of the vector computation is ready, the SPU then sets the VSJ bit. The VS will perform the conditional Jump instruction when it reaches the point where further execution requires the additional data.

The **SignalS** instruction performs the communications from the VS to the SPU. It sets the VSC in PSWO and interrupts the SPU.

Since the vector computation algorithm for robotic control is well-defined and the execution sequence is very straightforward, these basic vector sequencing instructions are sufficient to perform proper control sequencing for the vector
computation. A more sophisticated instruction set is not necessary because it is difficult to write sophisticated software for a SIMD computing array.

5.5 RSP-to-RSP Communications

Data communications among RSPs is more complex than the data communications within an SSIMDA structure because executions among RSPs are not synchronized the same way RVPs are within an SSIMDA structure. This section suggests a way to achieve RSP-to-RSP communications through the cooperation of the SPU, CM, and I/O channels.

All communications between RSPs is between their CMs. Figure 21(a) shows the data communications between two RSPs connected by CH0's of the respective RSPs. The SPU first checks the I/O channel output ready-to-send bit. If it is set, the SPU executes an `out` instruction. The `out` instruction specifies the output channel number, starting CM address, number of words to be transferred, and packet ID. The channel clears the ready-to-send bit after it receives the `out` instruction to suspend further output requests. The data transfer starts after the receiving channel signals ready-to-receive. Upon completion of the data transfer, the sending channel sets the ready-to-send bit to enable channel for output again, and the receiving channel disables further input and interrupts the receiving SPU. The interrupt routine of the receiving SPU checks the packet ID and determines further actions. It then executes an `in` instruction to enable the channel for input again.

Figure 21(b) is the format of the data packet. The data is 4-bit wide to match the width of the channel. The header includes the destination address of the data and a packet ID. It is followed by an arbitrary number of data (multiples of eight).
a) RSP-to-RSP communication scheme

1). SPU-getpsw1: checks ch0 status ready-to-send = 1?
2). SPU-out ch0, addr, ID, #wds initiates output data packet at ch0
3). Ch0- resets ch0 status ready-to-send = 0
4). Ch0- checks RSP2 ch0 status ready-to-receive = 1?
5). Ch0- sends data packet
6). Ch0- sets ch0 status ready-to-send = 1

1). Ch0- checks ch0 status ready-to-receive = 1 ?
2) Ch0- receives data packet
3). Ch0- interrupts SPU
2). SPU- processes data
2). SPU-in ch0 sets ch0 status ready-to-send = 1

b) RSP communication data packet format

<table>
<thead>
<tr>
<th>CM address[11:8]</th>
<th>data packet ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM address[7:4]</td>
<td>data</td>
</tr>
<tr>
<td>CM address[3:0]</td>
<td></td>
</tr>
<tr>
<td>data packet ID</td>
<td>(must be multiples of 8)</td>
</tr>
<tr>
<td>data</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 21: RSP-to-RSP data packet communication.
The destination address is the same as the starting source address; therefore, the data will occupy the same location in the CMs of both RSPs.

Looking at the system from the global aspect, the multiprocessor system behaves like a data-driven system because the input data packets trigger the SPU activities. This is very similar to the behavior of the real-time robotic control system in which all activities are triggered by the inputs from the high level command system or the controlled robot. The close match between the multiprocessor architecture and the target application is the most desirable feature for any special architectural design.

So far, a system architecture that maps the application in both the computation and communications areas has been presented. The next section further illustrates how they work together in parallel.

5.6 Parallel Execution of Scalar Computations, Vector Computations, and Communications

The architectural design of the multiprocessor system employs a high degree of distributed processing to reduce dependency. At the top level, RSPs are triggered by input data to perform certain tasks. They, in turn, generate output data to trigger further computations in the receiving RSPs until they reach the control output stage. There is no central controller to schedule and synchronize the activities among RSPs. This is the most natural form of asynchronous parallelism.

Within the SSIMDA, the vector computations and the scalar computations are executed in parallel. Figure 22 shows how the SPU coordinates the parallel execution among the scalar, vector, and global data communications activities. The SPU initiates a vector computation by executing a \texttt{forkV} instruction. The VS interrupts the SPU and informs the SPU its status via \texttt{signalS} instruction. The
SPU may disable the interrupt until it completes certain critical tasks.

The SPU may initiate *out* instructions to send data packets through the four I/O channels. The output channels carry out the actual data transfers and respond by setting the channel ready-to-send bit upon completion of the data transfers. Data input is also handled by the channel without the assistance of the SPU. The input channel notifies the SPU by interrupting the SPU and setting the corresponding fields in PSW1.

The parallel processing scheme presented in this section employs the minimal amount of interference by the SPU to achieve great amount of parallelism. This helps to prevent the SPU from becoming the bottleneck and limits the throughput of the layered architecture.

### 5.7 Summary

This chapter has presented the architecture design of the Robotic Scalar Processor. The architecture consists of a $512 \times 32$ dual-port Common Memory (CM), a Scalar Processor Unit (SPU), a Vector Sequencer (VS), a Vector data Channel, and four I/O channels. Common Memory is the key to the success of data communications between the various devices because it is involved in every data transfer. The SPU is the ultimate controller of the entire Synchronized SIMD Array, and is also a critical functional unit.

The RSP architectural design approach is very similar to that of the RVP architectural design; to pack more function units onto one chip, and to simplify each function unit to minimize its silicon area consumption. Figure 23 is a possible floor plan for the proposed RSP architecture. The CM occupies about 55 percent of the chip area, and the SPU consumes about 25 percent of the chip area. The
Figure 22: Overlapped scalar, vector, and communication operations.
architecture may be realized on a die of size $7900 \times 9200 \lambda$ with a 1.25 $\mu m$ CMOS fabrication technology.

The next chapter will present the VLSI design and layout of the RVP. Chapter 7 will present the implementation of the Inverse Plant Plus Jacobian Robotic Control algorithm using RSPs and RVPs.
Figure 23: Proposed Robotic Scalar Processor floor plan.
CHAPTER VI

THE ROBOTIC VECTOR PROCESSOR DESIGN AND VLSI LAYOUT IN CMOS

6.1 Introduction

This chapter presents the design and layout of the Robotic Vector Processor (RVP) chip. The general description of the major functional blocks of the RVP will be presented in this chapter, while the detailed circuit designs are discussed in Appendix B.

Figure 8 is the functional block diagram of the RVP. The processor includes five major functional blocks, namely, the Register File (RF), the Floating Point Adder (FPA), the Floating Point Multiplier (FPM), the Shift and Broadcast Network (SBN), and the I/O channel (IO).

The timing of the RVP is described in Section 6.2. The controller design is presented in Section 6.3. Sections 6.4 through 6.8 describe the design of each of the five functional blocks. Section 6.9 presents the VLSI layout, simulation, and testing of the RVP chip.

6.2 RVP Clocking Scheme

The clocking scheme of the RVP is determined by the timing requirements of the data-path. The timing of the control path is not critical because of the simple RVP controller circuit. The timing requirements of the data-path are quite
straightforward because the majority of circuits are designed with static CMOS logic. A two-phase non-overlapping clock scheme is sufficient to pace the execution of all of the functional blocks except the register file. The register file requires a four-phase clock to reduce the time wasted on precharge. Section 6.4 will discuss the timing requirements of the register file in further detail. In the RVP clock scheme detailed in Figure 24, $clk_{in}$ is the input external clock. The clock scheme may be generated by a two-phase clock generator, a one-bit counter, and a two-bit Johnson Counter. The targeted processor cycle for the RVP is 125 ns.

6.3 RVP Control

The RVP controller circuit is simple for the following reasons:

1. The RVP is a SIMD machine in which all three FPPs are controlled by one control circuit.

2. The controls of multi-cycle FPA, FPM, and I/O operations are performed by the controller of these individual units. The RVP controller uses only one cycle to initiate these operations.

3. The RVP does not have an on chip sequencer nor a large microcode memory. All control signals are generated by decoding the externally supplied instructions.

Figure 25 shows the control signals generated by the RVP controller. These control signals may be easily implemented with several small PLAs.

6.4 Dual Port Register File

The success of the FPP’s dual bus architecture relies heavily on the performance of the on-chip dual port register files. Each FPP has 64 addressable registers
Figure 24: RVP clock scheme.
Figure 25: RVP control signals.
with 32 bits per word. Of the 64 registers, 56 are dual-port registers made up of dual-port register cells, while the remaining 8 are the temporary latches located in the FPM, FPA, SBNA, SBNB, CHA, and CHB as shown in Figure 8.

Figure 26 is the circuit diagram of the register file. The register cell is made up of six transistors, four for the inverter pairs and two for the word selection. The output states of the inverter pairs, Cella and Cellb, are always complementary. Figure 26 shows that voltages on bitlines Bitbs are the voltage complements of the voltages on bitlines Bitas. Due to the fact that there are 5376 register cells in these register files, compactness in cell design is obviously desirable. A second benefit of a reduction in cell size is a corresponding reduction in parasitic capacitance of the bitlines, and thus a reduction in register delays.

The CMOS P-well double metal fabrication process includes several layers, metal 1 (M1), metal 2 (M2), polysilicon, n-diffusion, and p-diffusion. M1 and M2, the two metal layers, are often used for routing of signals which travel over long distances. The bitlines are routed horizontally on M2 and the wordlines are routed vertically on M1 layer. Bitlines are connected to the cell through word select pass transistors. Bitlines are precharged by clk1 at the start of every instruction cycle. The wordlines are discharged by clk4 at the end of the cycle. Figure 27 is the timing of the register file operations.

The register file read operation will cause either no change on the bitlines (if the connected CellAs or CellBs are high), or the discharge of the bitline (if the connected CellAs or CellBs are low). The word select transistor in this second case acts as a voltage divider. If the word select transistor is too wide, the voltage drop across the transistor will be too small, which may improperly allow the charges on the bitline to overwrite the state of the cell during read operation. A narrower
Figure 26: RVP dual-read, single write register file.
Figure 27: RVP register file timing.
transistor will have a greater voltage drop and avoid the improper overwrite; however, it will also increase the delay of the register file because of the smaller amount of current flowing through the transistor.

In order to reduce the read disturbance caused by the charge on the bitlines, the precharge circuit uses n-type transistors to pull the bitlines up to 4.0 volts instead of using p-type transistors which would pull the bitlines up to 5.0 volts. This has two advantages. First, the lower precharge voltage reduces the charge on the bitline. Second, the smaller voltage swing from 4.0 volts to zero volts reduces both the register file delay and the power consumption.

The sizes of the transistors of the register cell are determined by examining the read and write operations. Since the bitlines are precharged, the pull-up transistors in the cell need not be large. The aspect ratio of the p-transistors is chosen to be 3/7 (width/length). On the other hand, the pull-down transistors should be strong for fast response. Larger pull-down transistors may also reduce the bitline disturbance during read operation. The sizes of pull-down transistors are set to be 6/2. The word transistor size is chosen to be 4/2.

During write operations, the bitlines are driven with complementary signals. The bitlines are driven by both the cell transistors and the write drivers of the register file. The states of the cell will remain either unchanged, or will be altered by the write drivers. In the second case, the write drivers will first overcome the resistance of the pull-up transistor of the cell and pull that bitline from 4V to 0V. This triggers the change of the state of the cell.

6.5 Floating Point Adder

The design goal of the floating point adder/subtractor is to complete a single precision floating point add/subtract in three processor cycles. Subsection
3.2.2 has described the floating point format used for the floating point processor design. The 32-bit floating point format contains a sign bit, 8 biased exponent bits, and 23 (plus one implicit) mantissa bits. The exponent and mantissa bits are forced to 1 under overflow conditions, and are forced to 0 under underflow conditions. The operation of floating subtraction is implemented via addition using a 2's-complemented subtrahend. The algorithm for the floating point addition/subtraction is as follows:

1. PRENORMALIZATION - Align the two operands by shifting the mantissa of the smaller operand to the right by the amount of the difference of the exponents.

2. ADD/SUBTRACT - Add/subtract the aligned mantissas.

3. POSTNORMALIZATION - Shift left the result of Step 2 and subtract the shift amount from the exponent. The shift amount is equal to the number of leading zeros.

4. OVERFLOW/UNDERFLOW - Force all exponent and mantissa bits to 1 when overflow occurs. Force these bits to 0 if underflow occurs. This makes the maximum representable value to be about $\pm 2^{128}$ and the minimum number be $\pm 2^{-127}$.

Figure 28 is the block diagram of the FPA, which describes the design from a functional point of view. The details of the circuit design are presented in Appendix B.1.

The input operands are stored at the input master-slave latches after the first instruction cycle. The prenormalization is performed during the next cycle. The 31-bit comparator first compares the exponents and mantissas of these two
Figure 28: FPA block diagram.
operands and generates an \textit{AgtB} signal for the sign, exponent, and mantissa switch units. After the operands are switched, the mantissa from the smaller operand is used as the input of the five fixed-step alignment shifters. The larger exponent is subtracted by the smaller exponent and produces an 8-bit difference. The five LS bits of the difference are used to control the five shifters of step sizes 1, 2, 4, 8, and 16 bits. The two lines that controls the shifting of 8 and 16 bits are forced to one whenever the any one of the three MS bits is one (which means the difference exceeds 24). The shifting can take place as soon as the LS bit is calculated. The generation of the shifting control signal may overlap with the shifting operation. The detailed design of the comparator, switch units, exponent subtractor, and the alignment shifters is described in in Appendix B.1.3 through B.1.6.

The alignment circuit can also be implemented by a 24-bit barrel shift. However, the delay of the barrel shifter is greater because the shifting of the mantissa cannot take place until the barrel shifter decodes the five shift control lines into 25 control lines required by the barrel shifter to control the shifting operation.

At the end of the prenormalization cycle, the mantissa of the smaller operand is aligned and fed to input B of the 24-bit carry propagation adder. The next step is to add or subtract the mantissa \((M_{\text{max}} \pm M_{\text{min}})\). The operation performed by the carry propagation adder depends upon the signs of the operands and the \textit{Sub-Add} control signal. The carry propagation adder is implemented with the Manchester carry chain with carry lookahead. The following subsection describes the design of Manchester carry chain in detail. Appendix B.1.8 describes the adder circuit producing the propagate, generate, and sum signals.

The third step includes the leading zeros removal, exponent adjustment, and overflow/underflow handling. One way to perform the leading zeros detection is to use a PLA to implement the 24-bit leading zeros encoder [40]. However, this
would require a substantial amount of silicon area because this PLA uses at least 24 Min-terms. It also creates problems with the floor plan because of its dissimilar geometry and power rail arrangement.

The leading zeros removal scheme used in this processor design is similar to that of the alignment circuitry. There are five stages in removing the leading zeros, each of which employs a pseudo-nMOS NOR gate to detect the existence of a fixed number of leading zeros, such as 16, 8, 4, 2, and 1. The mantissa may be shifted left by that fixed amount if the number of leading zeros exceeds it. The design of the leading zeros detector is described in Appendix B.1.9.

At the end of the leading zero removal circuitry, all leading zeros will be removed. The number of leading zeros is encoded by five bits; this number is then subtracted from the exponent. The leading bit of the output should be the implicit bit "1". If the leading implicit bit is a "0", it means that the mantissa is zero and an underflow condition exists. This bit is then fed to the underflow control unit.

The exponent is reduced by the number of zeros removed. After that, the overflow/underflow unit checks for the abnormal conditions. Overflow occurs when the resultant exponent exceeds 255. Underflow occurs if the exponent is negative or if the mantissa is zero. The overflow and underflow signals are fed to the clamping circuit which forces all exponent and mantissa bits to 1 when overflow exists and forces all bits to 0 for underflow. The sign bit is not affected by the overflow/underflow condition. Appendix B.1.10 describes the circuit design of the overflow/underflow handler.

Finally, the result is stored at the output latches. The output register will be accessed by later instructions as register 6 of the 64 addressable registers.
6.5.1 Manchester Carry Propagation Chain

For an arithmetic operation such as addition or comparison, the calculation of the more significant bit depends upon the carry from the lower significant bit. This carry starts from the least significant bit, ripples through every higher bit, and reaches the most significant bit. The carry propagation delay becomes very significant as the word size increases. The longest carry propagation chain in the FPA is 31 bits employed by the 31-bit comparator. The FPA also employs a 24-bit carry propagation adder for the addition/subtraction of mantissas, and three carry propagation adders of length 8 to 9 bits for the exponents. The performance of the carry propagation chain is critical to the performance of the FPA.

Generally, the adder or comparator first translates the input operands into propagations (p's) and generations (g's); then uses these signals as the input of the carry propagation chain. For each stage, the carry input is first ANDed with the propagate signal and then ORed with the generate signal. Using a static logic implementation, the carry propagation delay for each stage is the delay of an AND gate plus an OR gate. For an average gate delay of 3 ns, a total delay of more than 180 ns could result for a 31-bit carry chain.

A special dynamic Manchester carry chain with partial carry lookahead is used in this design to reduce the delay with a minimal increase in size. This design is effectively a trade-off between gate requirements and carry delay.

Figure 29 is the circuit diagram of a 4-bit dynamic Manchester carry chain with carry lookahead. The input carry is first inverted, propagated through four n-type pass transistors, and inverted again at the output. The nodes on the carry chain are initially precharged, then selectively discharged. The discharging ripples through the pass transistors controlled by the propagate signals. The nodes are
also discharged by the transistors with the source connected to ground and the
gate controlled by the generate signals. The choice of negative logic is due to the
use of n-transistors for the pass transistors.

The choice of 4 bits in each segment is based upon two factors. First, as can
be seen in Figure 30, the Manchester carry chain is a series of pass transistors. The
behavior of such a carry chain is equivalent to that of a transmission line where
the channel resistance and the source/drain capacitance of the pass transistors
dominate the carry line parasitics. As the number of bits increases, the RC product
increases drastically. The insertion of inverter buffers in the carry chain effectively
limits the RC product and thus linearizes the overall size-delay relationship.

The second factor, the body effect, is due to the fact that the source of the
transistor has a higher voltage than the substrate [41,42], caused by the cascading
pass transistors. This effect further degrades the performance of the carry chain.
The SPICE simulation for the Manchester carry chain has shown that the gate
delay of the pass transistors increases from 3ns for the first stage to 5ns for the 4th
stage. Furthermore, the carry signal degrades as it propagates through the stages.

The total carry propagation delay of the 4-bit dynamic Manchester carry
chain is about 24 ns. A 31-bit carry chain would yield an unacceptable delay of
about 180ns. The partial carry lookahead scheme shown in Figure 29 is designed
to reduce the carry propagation delay. The pseudo nMOS NOR gate produces
the block carry propagation signal, p4, which controls the only pass transistor
of a separate carry propagation chain. This separate carry chain is dedicated to
propagate the carry input to the carry output. This effectively reduces the number
of cascading pass transistors to just one with a four-bit propagation delay of only
7ns.
Figure 29: Four-bit Manchester carry propagation chain with partial carry lookahead.
Contrary to other carry lookahead schemes, the Manchester carry lookahead scheme allows the size of the carry lookahead propagation block to be extended without additional area and delay penalties. Carry lookahead blocks of 8 bits are used in the 31-bit carry propagation adder of the FPM.

The Manchester carry chain is the only dynamic circuit employed in the RVP design. The timing between the precharge and the operand inputs is critical because the charges on the carry chain must not be discharged improperly.

6.5.2 Control Timing and Control Signal Generation

The FPA does not have internal pipeline registers. However, due to the usage of dynamic Manchester carry chains in several portions of the FPA, the FPA needs to have a control signal generator to ensure the proper execution of the FPA. The control timing signals are shown in Figure 30. Signal fpa.ph2bar is used to precharge the comparator carry chain during the latching of input data. Signal s1.2bar is used to precharge the exponent subtractor for generation of alignment shifters control signals. Signal s1.1bar is to precharge the exponent and mantissa adder, while s3.2bar precharges the exponent subtractor for the postnormalization. Signal s4.ph1 is used to load the result into the output latch at the end of the floating point addition/subtraction.

The controller is basically a ring counter consisting of five master-slave latches using either φ1 or φ2 as control clocks. When fpa goes high to start the add/subtraction operation, the controller propagates the high signal a step at a time to the next latch. Even though the ring counter structure uses more latches to represent fewer states, it actually consumes less silicon area by eliminating the irregular combinational circuit between the latches.
Figure 30: FPA timing signals.
6.6 Floating Point Multiplier

The design goal of the FPM is to complete a single precision floating point multiplication in 6 cycles. In addition, the RVP architecture requires the FPM to be pipelined every four cycles. This section describes the functional design of the FPM. The circuit design details are included in Appendix B.2.

The algorithm for the floating point multiplication may be described as follows:

1. Perform 24-bit fixed point multiplication of the mantissas of the two operands.

2. Add the two exponents, and correct for the bias.

3. Normalize the result of the mantissa multiplication. Mantissa shifting and exponent incrementing may be necessary.

4. Handle overflow and underflow conditions.

Figure 31 is the block diagram of the FPM designed for the RVP. The input MS latches latch the operands during the first cycle. In the next four cycles, the exponents are added, and the mantissas are multiplied. The 31-bit carry propagation addition, exponent adjustment, and the overflow/underflow are performed in the next cycle. The result is then latched into the output register.

The mantissa multiplication employs a pipelined recursive multiplication scheme which includes six rows of Carry-Save-Adders (CSA) and a 23-bit Carry Propagation Adder (CPA) [9]. The CSA arrays produce two 24-bit numbers after 4 iterations. These two numbers are then fed to the carry propagation adders to obtain the final product. The mantissa CPA is cascaded with the CPA for the exponent adjustment and creates a 32-bit CPA. The details of the CSA array design
Figure 31: FPM block diagram.
and the 32-bit Manchester carry propagation adder are described in the following subsections.

6.6.1 Recursive Partial Parallel CSA Multiplier

The mantissa multiplier is a 24-bit fixed point number multiplier. There are many schemes for the design of fixed point multipliers. The simplest is the sequential add and shift scheme. It uses the same adder n times to perform the addition of n rows of a partial product. For a 24-bit Manchester carry propagation adder with carry lookahead the delay is about 70ns for every 4 bits. The total multiplication delay is then 1680 ns which is much too long compared to the speed of the FPA.

On the other extreme, the fastest multiplier can be implemented with arrays of adders with each full adder corresponding to one partial product term tree. However, the size of these multipliers increases dramatically as the word width increases. A fully parallel 24-bit multiplier would require twenty-four 24-bit carry propagation adders or twenty-four 24-bit carry-save adders plus a 24-bit carry propagation adder. These adders consume too much precious silicon area and leave too little space for other functions. This leads to the choice of a partial parallel scheme trading off performance against the area consumption.

The multiplication scheme employed in this processor design is shown in Figure 32. The multiplier consists of six rows of 24-bit CSA, 46 master-slave latches, and a 23-bit carry propagation adder. The CSA element is a full adder with the carry flowing the same direction as the A and B inputs. The carry flows straight downward from the top array to the bottom, while the sum propagates diagonally toward the right from top to bottom. Thus, proceeding from top to bottom, the weighting of the bits in each successive row is twice those of the preceding row.
Figure 32: Carry-Save-adder multiplier.
In each cycle, the mantissa of operand B is shifted right by six bits. The six least significant bits are \textit{anded} with the mantissa of A to produce six rows of partial products as the inputs of the six CSA arrays. The execution of the six CSA arrays produces 6 least significant bits of the product, 23 carries, and 23 sums. The 23 carries and sums are latched by the 46 master-slave latches and are fed back to the carry-save adders at the next cycle. After four iterations, all 24 rows of partial products are processed. The result of the carry-save adders is the least significant 24 bits of the product plus 23 pairs of sums and carries. The sum of these two 23-bit numbers is the most significant 24 bits of the product. The 24 least significant bits are ignored. A rounding scheme for the mantissa is not implemented in this processor for the sake of design simplicity.

This recursive partial parallel CSA multiplier scheme is very flexible in terms of the choice of time delay and area consumption. If the number of CSA stages is 1 it is similar to the sequential add and shift multiplier. One the other extreme, when the number of CSA stages is 24, it becomes a full parallel multiplier yielding a 48 bit product. The choice of the number of stages depends upon the speed requirement of the multiplier and the available silicon area. In this processor, six stages are chosen because this is consistent with the 125 ns processor clock. The six CSA stages require four cycles to complete the mantissa multiplication. The four-cycle pipelining is in line with the architecture pipeline scheme of the RVP.

6.6.2 Exponent and Mantissa Manchester Carry Propagation Adder

The exponent is calculated by adding the exponents of the two operands. Since the exponents are represented with bias of 127, the resultant biased exponent is:
\[ E_c = E_a + E_b - 127 \]

where 127 is subtracted to correct for the double inclusion of a bias factor in the sum \( E_a + E_b \)

\[ = E_a + E_b + 1 - 128 \]

\[ = (E_a + E_b + 1) + (2's \text{ complement of } 128) \]

\[ = E_1 + 110000000 \]

where \( E_1 \) is \( E_a + E_b + 1 \)

The function is implemented with two carry propagation adders. The first adder implements \( E_a + E_b + 1 \), where the addition of 1 is implemented by setting the input carry, \( C_0 \), high. The first 8-bit adder generates a 9-bit number, \( E_c \), which is the 8-bit sums and the output carry, \( C_0 \). The second adder adds \( E_1 \) and 110000000 to adjust for the bias.

The first exponent adder is an 8-bit Manchester carry propagation adder. It consists of two serial 4-bit ripple-carry chains without carry lookahead. The second exponent CPA, a 9-bit carry propagation adder, adds the 2's complement of 128 from the 9-bit result of the first CPA. The input carry of the second adder is derived from the carry output of the mantissa CPA of the mantissa multiplier to account for mantissa normalization.

The output of the second exponent CPA is the exponent of the product subject to the overflow/underflow adjustment. Since the 9-bit exponent CPA takes the carry output from the 23-bit mantissa CPA, the two actually form a 32-bit CPA.

### 6.6.3 Postnormalization

There are two tasks involved in the postnormalization, the mantissa alignment and the overflow/underflow control. If the carry out of the 23-bit mantissa CPA output is a 1, the exponent will be incremented by 1 and the CPA outputs require
no shifting. On the other hand, if the carry out of the mantissa CPA is 0, the exponent will not be incremented but the mantissa will be shifted left by one bit.

The overflow/underflow control for the FPM is similar to that of the FPA with the exception that the mantissa will not cause underflow in the FPM operation.

6.6.4 Timing and Control Signals Generation

Figure 33 is the timing diagram of the FPM. The state variables S1, S4bar, S4ph2, and S6 are used to control the various parts of the FPM. The design objective is to design the circuit so that the propagation delays for various paths stay within the 125ns cycle time.

The control signal generation circuit is basically a six-DFF ring counter similar to that of the FPA. The assertion of the fpm signal triggers the generation circuit and the signal is rippled down the control latches. This scheme is simple and easy to implement. The output of the circuit is buffered to shorten the delay caused by the large capacitance load.

6.7 Shift and Broadcast Network

The dual SBNs are responsible for transferring the data among the FPPs so that proper operands are available at each FPP. The SBN, as shown in Figure 34, consists of three 32-bit registers. The three registers form a parallel circular shift network which may shift the 32-bit word forward, backward, or not at all.

Broadcast is performed by shifting the shift registers forward and setting R4/R5. One of the three shift busses, Sh-Bus-1, is connected to the data busses (BAs/BBs) of the three FPPs through 96 pass transistors. These pass transistors, controlled by R4/R5, serve as the broadcast registers which broadcast the value on Sh-bus-1 to all three FPPs. Through proper control of the shift register, SBN
Figure 33: FPM control timing diagram.
Figure 34: Shift and Broadcast Network block diagram.
and FPPs can be used to implement vector cross product, vector dot product, and matrix-vector multiplication. The detailed circuit design of the SBN shift register is presented in Appendix B.3.

6.8 I/O Channel

Each I/O channel consists of three 32-bit registers, R0/R1. The I/O channel is also a shift network similar to the Shift and Broadcast Network. Data in these registers are shifted forward in parallel. The 32-bit output of the top register are fed to the input of bidirectional I/O pads and the 32-bit output of the bidirectional pads is fed back to the bottom register. Figure 35 is the block diagram of the I/O channel.

During idle states, the I/O pads read the data on the external bus and then move the data to the bottom register. In other words, the I/O channel is defaulted to read from the external bus unless the channel is in output state. The RVP can issue instructions to read the data from the I/O channels at any time.

Figure 36 is the timing of the I/O operation. Each channel has a controller consisting of three D-FFs. There are four states for an output operation. The first state, out is true, is similar to LSR instruction except that the destinations are the R0 and R1 registers. The out signal propagates through the D-FFs during the next three cycles. The EN\textsubscript{out} is true for the next three cycles and the pads send the data from the top shift registers to the external bus. The detailed circuit design of the I/O channel is in Appendix B.4.

6.9 VLSI Layout, Simulation, Testing of RVP

This section presents the CMOS VLSI layout, simulation, and testing. The design file is submitted to MOS Implementation System (MOSIS) for fabrication.
Figure 35: I/O channel block diagram.
Figure 36: I/O channel control timing signals.
Currently, MOSIS supports 3.0 $\mu$m CMOS fabrication. It is expected that MOSIS will support 1.25 $\mu$m fabrication in the coming year. The layout is based upon MOSIS 3.0 to 1.25 $\mu$m scalable CMOS design rule. At the present time, the function blocks, such as the FPA and the FPM are fabricated with 3.0 $\mu$m technology. However, the complete RVP chip will be fabricated by the 1.25 $\mu$m CMOS technology.

The RVP contains about 90,000 transistors and occupies a silicon area of 7900 by 9200 $\mu$m$^2$. For a VLSI processor of such complexity, a powerful Computer Aided Design (CAD) is essential for both design layout and simulation. The first subsection briefly describes the CAD tools used for the VLSI layout and design simulation. The next subsection presents several CMOS circuit characteristics which have great influence on the design style of the VLSI circuit layout. From these characteristics, some guidelines for the VLSI layout design can be derived. The third subsection presents the template for the design of standard cells based upon the VLSI characteristics. The determination of the template is of most importance because it decides the dimensions, floor plan, signal routing, and speed performance of the processor. The template allows the designer to lay out a denser, faster, and more reliable circuit with less effort. The fourth and fifth subsections present the layout, simulation, and testing of the FPA and FPM.

6.9.1 Computer Aided Design Tools for VLSI Chips

This subsection briefly describes the CAD tools used for the VLSI layout and design simulation of the FPA and FPM. These tools were developed at UC Berkeley. The most frequently used programs are MAGIC, ESIM, CRYSTAL, and SPICE. In addition, The Berkeley CAD software includes functional design tools to speed up the implementation of random logic circuits.
MAGIC - Layout Editor

*MAGIC* is an interactive editor for VLSI layouts that runs under BSD 4.2 Unix. At the Ohio State University, *MAGIC* is run on a VAX 780 computer and SUN 3/160C workstation. However, the major part of the design effort was spent on the SUN workstation.

*MAGIC* has built-in knowledge of layout rules; the program checks for rule violations every time the layout is modified. *MAGIC* is based on the Mead-Conway [43] style of design. The simplified design rules [44] and circuit structures it uses makes it easier to design circuits.

Although *MAGIC* is a physical layout tool, its repertoire of commands [10], such as routing and plowing utilities, makes it very powerful and friendly. It also has a built-in program to extract layout into a file which may be used by other simulation tools.

ESIM - Switch Level Simulator

*ESIM* [45] is an event-driven switch level simulator for NMOS and CMOS transistor circuits. It uses logic levels, and models the transistors after switch with different strengths. The simulation determines the node voltage based upon charge sharing on the node, and pull up and pull down transistor strength.

During cell design, *ESIM* is used to verify the logic correctness of the layout. Since most of the circuits adapt the static CMOS design style, the majority of layout modifications are performed during this period. After each cell has gone through complete *ESIM* simulation, it is certain that the static portions of the circuit will function properly and the dynamic portions of the circuit will very likely function.
After the cells are connected together, ESIM is used to verify the logic of the resulting circuit. It is very successful in pointing out the logic errors of the resulting circuit. The response time of the ESIM simulation for circuit with more than 11,000 transistors is only several seconds. The fast simulation response, in addition to its ability to catch the logic design error, makes ESIM a vital tool during the design process.

**CRYSTAL - VLSI Timing Analyzer**

CRYSTAL [46] is a semi-interactive program for analyzing the timing characteristics of large integrated circuits. It estimates the speed of a circuit and prints out information about the critical paths.

Even though CRYSTAL uses a simplified RC circuit model and a transistor slope model for timing analysis, it is found that in most cases CRYSTAL consistently produces delay values very close to the results of SPICE simulations. When the circuit contains several cascaded transistors or transistors driving very large loads, CRYSTAL does not produce good results. Under these circumstances, the circuit needs to be simulated with SPICE.

Usually, a timing analysis is used to determine the relative delay between different paths in the circuit. Therefore, as long as CRYSTAL produces consistent delay values, it may be used in place of circuit simulations. In this way, CRYSTAL can be used to fine tune the circuit while SPICE simulations are run to obtain accurate final timing information of the already tuned circuit.

**SPICE - Circuit Simulator**

SPICE is a general-purpose circuit simulation program for nonlinear DC, nonlinear transient, and linear AC analysis. It reads in the circuit data and generates accurate signal waveform information. The computation time required by the
SPICE simulation increases exponentially with the number of transistors in the circuit. For this reason, the SPICE simulation is run only for circuits with fewer than 150 transistors. As discussed in the previous subsection, the SPICE simulation complements the CRYSTAL simulation very well. In this processor design, SPICE is used extensively on cells in the critical paths to obtain accurate timing information.

6.9.2 CMOS Circuit Characterization

This subsection briefly presents several CMOS circuit characteristics that have great influence on the design style of the RVP. Among those are: layer sheet resistance, layer capacitance, transistor switching delay, power consumption, and latch-up. Table 6.1 shows several process and device parameters of the MOSIS 3 μm P-Well CMOS fabrication services.

Layer Capacitance. The switching of a CMOS circuit involves charging and discharging of load capacitance. The size of the capacitance directly affects the circuit speed performance. Table 6.1 shows that diffusion has the largest capacitance per unit area in addition to a large peripheral junction capacitance. For a square of 4 μm per side, assuming the average n-diffusion junction capacitance is 0.45 fF/μm² and the average peripheral junction capacitance is 0.3 fF/μm, the average n-diffusion capacitance is about 54 times greater than the average M2 capacitance, 32 times greater than the average M1 capacitance, and 16 times greater than the average polysilicon capacitance. This leads to the guideline that no diffusion should be used for signal connection.

Layer Sheet Resistance. Polysilicon and diffusion have similar resistivity. The resistance of both layers are at least 300 times greater than that of M1 and 600 times greater than that of M2. The RC product of a connection wire determines
Table 2: Device parameters of the MOSIS 3 μm P-well CMOS fabrication process.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Descriptions</th>
<th>Units</th>
<th>N channel</th>
<th>P channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rho_{\text{diffusion}}$</td>
<td>Diff. Sheet R.</td>
<td>ohms / □</td>
<td>&lt;40</td>
<td>&lt;80</td>
</tr>
<tr>
<td>$\rho_{\text{poly}}$</td>
<td>Poly. Sheet R.</td>
<td>ohms / □</td>
<td>&lt;30</td>
<td>&lt;80</td>
</tr>
<tr>
<td>$\rho_{\text{well}}$</td>
<td>P-well Sheet R.</td>
<td>1 k ohms/ □</td>
<td>3.5 - 7.0</td>
<td></td>
</tr>
<tr>
<td>$\rho_{\text{metal 1}}$</td>
<td>M1 Sheet R.</td>
<td>ohms / □</td>
<td>&lt;0.1</td>
<td></td>
</tr>
<tr>
<td>$\rho_{\text{metal 2}}$</td>
<td>M2 Sheet R.</td>
<td>ohms / □</td>
<td>&lt;0.05</td>
<td></td>
</tr>
<tr>
<td>Cj</td>
<td>Diff. Junc. Cap.</td>
<td>$f F / \mu^2$</td>
<td>0.3 - 0.6</td>
<td>0.12 - 0.6</td>
</tr>
<tr>
<td>Cjp</td>
<td>Diff. Peri. Cap.</td>
<td>$f F / \mu$</td>
<td>&lt;0.6</td>
<td>&lt;0.6</td>
</tr>
<tr>
<td>C m2 m1</td>
<td>M2 over M1 Cap.</td>
<td>$f F / \mu^2$</td>
<td>0.031 - 0.045</td>
<td></td>
</tr>
<tr>
<td>C m2 poly</td>
<td>M2 over Poly. Cap.</td>
<td>$f F / \mu^2$</td>
<td>0.020 - 0.024</td>
<td></td>
</tr>
<tr>
<td>C m2 over field</td>
<td>M2 overfield Cap.</td>
<td>$f F / \mu^2$</td>
<td>0.012 - 0.016</td>
<td></td>
</tr>
<tr>
<td>C m1 poly/diff</td>
<td>M1 over P/D Cap.</td>
<td>$f F / \mu^2$</td>
<td>0.035 - 0.05</td>
<td></td>
</tr>
<tr>
<td>C m1 over field</td>
<td>M1 overfield Cap.</td>
<td>$f F / \mu^2$</td>
<td>0.020 - 0.027</td>
<td></td>
</tr>
<tr>
<td>C poly</td>
<td>Poly. Cap.</td>
<td>$f F / \mu^2$</td>
<td>0.036 - 0.055</td>
<td></td>
</tr>
<tr>
<td>R m2 m1 (3 μ x 3 μ)</td>
<td>Via contact R.</td>
<td>ohms</td>
<td>&lt;0.1</td>
<td></td>
</tr>
<tr>
<td>R m1 poly (3 μ x 3 μ)</td>
<td>M1/Poly cont. R.</td>
<td>ohms</td>
<td>&lt;10</td>
<td></td>
</tr>
<tr>
<td>R m1 diff (3 μ x 3 μ)</td>
<td>M1/Diff. cont. R.</td>
<td>ohms</td>
<td>&lt;5</td>
<td></td>
</tr>
</tbody>
</table>
the wire delay. The wire length of any layer should be short enough that it does not introduce any significant delay in addition to transistor gate delays. This leads to the guideline that polysilicon wire should not exceed 200 \( \lambda \) \[41\].

**Transistor Switching Delay.** A P-channel transistor takes twice as long to switch than an N-channel because of the lower mobility of holes. P-transistors in the critical path drastically increase the circuit delay, then the circuit design should contain a minimal number of P-transistors in the critical path. A CMOS NOR gate has serial P-transistors and thus poor pull-up speed. Increasing the P-transistor width and reducing its load capacitance can also improve the circuit performance.

**Power Consumption.** CMOS circuit has much less power dissipation than other processes because of its small static power consumption. However, if the clock rate increases, so does the dynamic power dissipation. In general, power consumption can be reduced by avoiding dynamic and pseudo-NMOS circuit design styles. It is also helpful to start the layout with smaller transistors and then increase only those transistors in the critical path.

### 6.9.3 Template for Standard Cell Design

The template for standard cell design of the RVP data-path is shown in Figure 37. The issues involved in the determination of the template include power rail routing, data signal routing, control signal routing, n-type and p-type transistor placement, global signal routing, local signal routing, capacitance and resistance reduction, latch-up prevention, and abutting arrangement.

The standard cell has a fixed vertical pitch of 92 \( \lambda \)s. The width of the cell is flexible depending upon the complexity of the circuit. Typically, the width of the cell is less than 150 \( \lambda \)s. The power wires are 12 \( \lambda \) wide and are routed horizontally.
Figure 37: Standard cell template.
on M2 layers. The width of the power wires is chosen so that the wires may provide sufficient power for all the cells abutted horizontally across the entire chip. The GND wire routed through the middle of the cell and the two Vdd power wires are routed at the top and the bottom of the cell. Vdd power lines are shared by the vertically abutted cells.

There are 56 λs of vertical spacing between the power rails. The spacing is divided into 14 horizontal tracks of 4 λ wide each. These tracks, labeled as D1 through D6, are used to route M2 wires. The eight tracks between these M2 wires are used to place transistors which use M1, diffusion, and polysilicon layers. Since the layers used between the adjacent tracks are different, each track serves as the spacing between its neighboring tracks. This eliminates the wasted area for the spacing requirements.

M2 wires are used to route data signals horizontally along the six tracks assigned to them. Since the M2 layer has very small resistance and parasitic capacitance, it can be used to carry large current over a long distance. It is not used for local signal routing because there is no contact between the M2 layer and other layers except the M1 layer [44].

Contacts are available only between M1 and other layers resulting in a great demand for M1. The careful usage of M1 is important if efficient layout is to be achieved. Control signals are routed vertically across the cell. Usually the control lines are used by the same cell, repeated many times vertically. This suggests that the control lines should be routed straight across the cell from the top to the bottom. Thus the M1 layer is used almost exclusively for the routing of control lines across the cell boundary with very few exceptions.

Local signal routing within the cell is usually handled by the polysilicon and the M1 layers. Both layers may route in any direction. The polysilicon layer is
preferred for local routing for two reasons. It has smaller spacing requirements than
the M1 layer, and it propagates the signal to the next transistor by simply crossing
the diffusion layer of that transistor. However, polysilicon wires very seldom travel
across the cell boundary due to the high sheet resistance of the polysilicon layer.
The design guideline is to avoid routing polysilicon lines over 200 \( \lambda \).

The diffusion layer is not used for interconnection since it has extremely large
capacitance relative to the other layers. The diffusion layer should be used only to
create transistors. The source and the drain of a given transistors are usually on
the same track. If the next transistor uses the same tracks, it is very possible that
the two adjacent transistors may share the same source of drain diffusion. This
not only saves the silicon area, but also reduces the diffusion capacitance which
directly translates into shorter circuit delay.

Out of the fourteen horizontal tracks, six are reserved for M2 wires. The
remaining eight tracks are available for the placement of transistors. The p-
transistors are placed primarily along Vdd wires in tracks T1 and T8. On the
other hand, n-transistors are placed along GND wires in tracks T4 and T5. Both
n-type and p-type transistors can be placed in tracks T2, T3, T6, and T7. How­
ever, the spacing necessary between n-type and p-type transistors prohibits the
placement of transistor of opposite types on adjacent tracks. In other words, two
of the four remaining tracks must be left empty.

N-type transistors are built on p-wells. These p-wells have to be well grounded
if the N-transistors are to be expected to function properly. For this reason, all
p-wells are laid along the GND wires with p-Well contacts to the GND placed
wherever space is available. This creates long well grounded p-well strips along
the GND wires. The n-type substrate is divided by the p-well strips into strips
along Vdd lines. Substrate contacts to Vdd are also placed along the strips to
create good substrate voltage at 5 volts. This p-well arrangement greatly reduces the chances of latch-up.

6.9.4 RVP Floor Planning

Figure 38 is the floor plan of the RVP based upon the estimated dimensions of each individual functional block. The RVP aims at MOSIS 1.25 µm bulk CMOS P-well fabrication as the target technology. The chip will be implemented on a die of size 7.9mm by 9.2 mm.

The floor plan determines the placement and routing of the global signals. It is important to have a good floor plan, since it directly effects the signal delay characteristics and the silicon real estate occupied by the communication circuits. The determination of the floor plan requires a knowledge of the circuit complexity and the dimensions of the standard cell. Because there are many possible ways to lay out the circuit, the final floor plan is the result of many iterations.

The FPPs are stacked vertically with the processor control circuit placed at the top. This placement simplifies the routing of the control signals which are generated by the control circuit on the top and routed vertically through the FPPs. This arrangement also reduces the routing area of the Shift Network.

6.9.5 FPA Layout, Simulation, and Testing

Layout

Figure 39 is the floor plan of the FPA. The dimensions of the chip are 2652 λ x 1187 λ. The data signals are routed from left to right. Control signals are routed from top to bottom.

The data busses BA and BB are routed through the entire FPA on the M2 layer. The FPA outputs are connected to BB through pass transistors controlled
Figure 38: RVP floor plan.

136
Figure 39: FPA floor plan.
by word select line WB2.

Cell layout considers several factors. First, it should minimize the width of the cell since the height is fixed. A narrower cell reflects a smaller area consumption and also a shorter data path which results in smaller parasitics. Second, the placement of the input and output signals is vital to reduce the length of the data path. Third, the cell has to be abuttable vertically since most cells are used repeatedly for several bits. Vertical abutting between different cells is sometimes also necessary.

Generally, the global control signals in the RVP travel longer distances than the data signals because they need to travel across the entire die. The generation of the control signals is critical to the performance of the processor. In all cases, the control signals travel vertically with little diffusion overlap.

The FPA employs static logic design in all but the Manchester carry chain circuitry. No cascading pass transistors are used other than in the precharged Manchester carry chain to ensure good signals and feedback isolation.

The FPA area consumption is very much dependent upon the placement and routing of the cells. The mantissa data path requires more area than the exponent. The control timing generator is placed in the free space of the exponent circuit layout area.

Simulation

Each cell is thoroughly tested with ESIM to verify the logic correctness. Normally, these errors are due to layout mistakes, such as when a M1 layer accidentally shorts the signal to GND. Sometimes ESIM flags an error condition on pass transistors because load states are unknown. While these may not be actual logic errors, to be cautious, cells such as the input and output latches with pass transistors
connected to large loads, are buffered by preceding inverters. This removes ESIM simulation errors.

The ESIM simulation provides such quick function verification that it reduces the cell design time drastically. As cells are composed into large function blocks, ESIM is also used to verify the switching functions of these blocks. This is where ESIM proves most useful. With proper test vectors, ESIM can detect almost all design errors except those caused by timing delays.

After the cells have been checked for correct logic, the CRYSTAL program is used to find the critical timing paths of these cells. CRYSTAL finds the critical paths delays. It provides quick feedback to the designer about the approximate timing performance of the layout, and allows the designer to modify the layout quickly. The timing information obtained from the CRYSTAL simulations are generally quite accurate.

Finally, most cells are simulated with the SPICE program to obtain the analog information about the circuit, such as the timing and voltage behavior. SPICE is most useful in testing certain crucial timing requirements such as the set-up time of the latches or the timing of precharge. Sometimes, such as the simulations for control MS latches, loads are added artificially to simulate the circuit performance for wider range of load. SPICE simulation is conducted on input MS latches, control MS latches, Manchester carry chains, and cells whose behavior was most crucial to the overall performance of the FPA. SPICE simulation was not used for cells with more than 150 transistors due to the tremendous amount of computation time required.

Figure 40 shows the critical path of the FPA. Simulation has shown that the FPA is capable of calculating a floating point add/subtraction in three 125 ns cycles after the the data is loaded into the input latches.
Figure 40: FPA critical path.
Testing

The FPA was submitted to MOSIS for fabrication. The test chip was fabricated and thirteen chips were returned after three months.

Figure 41 is the block diagram of the FPA test chip. This chip is laid out on a die of size 6800 times 6900 $\mu^2$ in a 64-pin package. The input and output data are multiplexed so that only 16 pins are required for the input data and another 16 pins for the output data. Several control signals are required to handle the multiplexing. Registers $R_{\text{ain}}$ and $R_{\text{bin}}$ hold the input test operands for the FPA. Register $R_{\text{bout}}$, the output latch of the FPA, holds the result of the floating point addition/subtraction. $R_{\text{ain}}$ drives bus BA constantly. $R_{\text{bin}}$ drives BB when $I/O = 1$, and $R_{\text{bout}}$ drives BB otherwise. $A/B$ and $H/L$ control both the input and output multiplexers. The $I/O$ signal determines whether the pins display the actual input data or the data stored in the output latch of the FPA. Figure 42 is the photograph of the FPA test chip.

All of the global signals on the RVP, such as the two-phase clocks, $\phi_1$ and $\phi_2$, are fed from outside sources. Several internal FPA signals are connected to the output pads for better testibility, such as the control timing and postnormalization shifter control signals.

The FPA chip was tested using the Tektronix 9100 DAS test station with thirty-two 40 ns pattern generators and thirty-two 100 ns data acquisition probes. The test chip was mounted on a 64-pin zero-insertion-force socket whose pins were all pulled-up by 62,000 $\Omega$ resistors.

Figure 43 is a sample program for the test pattern generation of the DAS. Input data are first latched into $R_{\text{ain}}$ and $R_{\text{bin}}$ registers. The execution of the FPA is followed by asserting $fpa$ high for one full cycle. Three cycles later, the
Figure 41: FPA test chip block diagram.
Figure 42: Photograph of FPA test chip.
data is available at the output latch, $R_{\text{bout}}$.

The FPA chip has been tested by 22 different input patterns. It has been tested for the addition and subtraction of numbers having the same or opposite signs, and overflow or underflow handling. The preliminary results have shown that the FPA chip functions correctly on each one of these tests for DAS clock rates up to 25 Mhz. Since the input clocks are generated by driving the pattern generation probes high and low alternatively, the test program is able to generate two-phase input clocks with an 80 ns period to the FPA chip. This is 45 ns faster than the SPICE simulated delay. The reduced delay may be caused by the worst case parameters used in the SPICE simulation. The excellent speed performance of the FPA is also an indication of a good layout strategy because cells are laid-out in a rather straightforward fashion according to the guidelines set earlier in this section.

6.9.6 FPM Layout, Simulation, and Testing

Layout

Figure 44 is the floor plan of the FPM layout. The actual dimensions of the layout are 2652 $\lambda \times 1234 \lambda$. This is equivalent to 3978 $\mu m \times 1851 \mu m$ in 3.0 $\mu m$ fabrication technology and 2121.6 $\mu m \times 987.2 \mu m$ for 1.25 $\mu m$ fabrication technology. The data and control signals are routed in the same way as in the FPA.

The CSA adder cell and the CSA dff latch cell are the most difficult to lay out because of the limited tracks available for data routing within these two cells. The sum and carry feedback paths, and the global data busses BA and BB occupy four of the six M2 tracks leaving two tracks available for routing tracks. In the case of the CSA cell layout, polysilicon paths are routed underneath the power rails to
Clock = 40 ns

Figure 43: FPA test program using Tektronix DAS 9100.
Figure 44: FPM floor plan.
provide the required extra routing. Generally speaking, cell layout is not difficult except when reduction of the horizontal dimension of the cell is attempted.

**Simulation and Evaluation**

The floating point multiplier has two major components, the six by 24 carry-save-adder array and the carry propagation adder. All circuits except the Manchester carry propagation chain are designed with static CMOS logic to simplify the control timing.

Each cell was thoroughly tested with *ESIM* to verify the logic correctness. *CRYSTAL* was then used to quickly find the critical paths of each cell. *SPICE* was extensively used to obtain timing and signal voltage information on all critical elements of the FPM.

Figure 45 shows two critical paths of the FPM circuit. Path 1 is the critical path of the mantissa carry-save-adder (CSA) multiplier. Figure 46 is the delay path of the CSA Array. This path propagates through $\overline{bbar} \rightarrow \text{sum} \rightarrow \text{sum} \rightarrow \text{carry} \rightarrow \text{sum} \rightarrow \text{carry} \rightarrow \text{sum}$. The total delay is 107ns. For a 125ns cycle time, this allows a margin 18 ns.

Figure 47 shows the delay path of the 32-bit carry propagation adder (CPA) and the overflow/underflow circuit. The \text{sum} and \text{carry} output of the MSB is then used to generate overflow and underflow signals. These signals are used to clamp the mantissa and exponent outputs. The total delay of this path is 96ns. Since the output is not used until 1/4 cycle into the next cycle, this path has a 60ns margin and is not nearly as critical as the CSA multiplier path.
Figure 45: FPM critical path.
Figure 46: Timing delay of 31-bit Manchester partial carry lookahead adder, alignment, and overflow/underflow control circuits.
D = A * B
S_i = D xor (C_i xor S_i)
C_i = D C_i + D S_i + C_i S_i

Figure 47: Carry-Save-Adder array critical path.
Evaluation of FPM Pipeline After Design

The SPICE simulation shows a possible cycle time of 107ns for the FPM. The FPM pipeline allows the FPM to execute a floating point multiplication every four cycles. If a pipeline with fewer cycles is desired, the number of carry-save adders in the CSA multiplier would have to be increased. The cycle time may have to be increased to allow enough time for the signal propagation through the CSA array. It can be seen that if two more rows of CSA adders are included, which increases the number of CSA adders in the CSA array to 8, the three cycle pipeline FPM would have a cycle time of 130ns. The determination of the FPM pipeline scheme depends heavily on the resultant cycle time, the RVP architecture, and the data dependency within the application.

The use of the pipelined recursive CSA multiplier provides the flexibility to tailor the FPM pipeline scheme to match the RVP pipelining scheme, which allows the RVP bus to be utilized efficiently.

The carry lookahead scheme used in the CPA uses about 15 percent of additional silicon area to implement a fast carry chain which propagates carry through for any number of bits in about the same time, 7ns. This carry lookahead scheme eliminates the 32-bit CPA and overflow/underflow clamping from the FPM from the list of critical paths.

This cycle time would be reduced further when the design is fabricated with 1.25 µm CMOS technology. The λ value will be reduced from 1.5 µm in the 3.0 µm technology to 0.8 µm. Based upon the first-order MOS scaling theory [41], the delay scaling factor is 0.53. This means that the FPM cycle time may be reduced from 80 ns to 43 ns when the FPM is fabricated with the targeted 1.25 µm technology.
Testing

The FPM was has been submitted to MOSIS for fabrication and is currently being fabricated. The layout of the FPM test chip is similar to the FPA test chip. Figures 48 is the block diagram of the FPM test chip.

All of the global signals on the RVP, such as the two-phase clocks, $\phi_1$ and $\phi_2$, are fed from outside sources. Several internal signals of the FPM are connected to the output pads for better testability, such as the control timing and postnormalization shifter control signals.

6.10 Summary

In this chapter, the circuit design of the RVP has been described. The FPA, FPA, and register cell has been laid-out and simulated. Both FPA and FPM have been submitted to MOSIS for fabrication. Thirteen FPA chips have been received and tested to function for 80 ns cycle time.
Figure 48: FPM test chip block diagram.
CHAPTER VII

PARALLEL IMPLEMENTATION OF THE ROBOTIC CONTROL APPLICATIONS

7.1 Introduction

In Chapter 3, a layered restructurable multiprocessor architecture consisting of two VLSI chips, a RSP and a RVP, was presented. This chapter describes the implementation of the Inverse Plant Plus Jacobian Robotic Control Algorithm using the restructurable architecture. The RSP and the RVP will be used as the building blocks to construct various computing structures to optimize the control performance.

Section 7.2 presents the implementation of basic vector operations using the Scalar Processor Unit of the RSP. Section 7.3 presents the implementation of basic vector operations with the RVP. These will be used as the fundamental operations to implement the larger tasks.

The results of Sections 7.2 and 7.3 are used in the following sections to pursue suitable architectures for the application. Section 7.4 presents several synchronized control schemes using one RSP and multiple RVPs. Section 7.5 presents three multirate implementation schemes using two, three, and four RSPs.
7.2 Basic Floating Point Operations Implemented by RSP

This section presents the parallel and pipelined execution timing of basic floating point operations implemented by the Scalar Processor Unit (SPU) of the RSP. The SPU contains an FPP for floating point operations and an ALU for integer operations. Only floating point operations are evaluated here because they make up the majority of the robotic control computation.

As was presented in Section 5.4, the SPU is a dual-bus processor containing an FPM, an FPA, an ALU, and an RF. Each RSP instruction cycle time is equal to two RVP instruction cycle times. In order to compare the number of operations performed by the RSP on the same basis as the RVP, all execution cycles referred to in the remainder of this chapter are RVP instruction cycles. Therefore, the FPA operation takes four cycles, and the pipelined FPM operation takes six cycles with a four-cycle pipeline latency. The result of a floating point operation is latched in the output register of the FPA or FPM. The succeeding FPA or FPM writes the results back to the RF. Following the completion of the last floating point operation, an FPA or FPM operation will store the results of the output latch back into the RF.

A certain amount of overlap between successive operations is possible. Overlapping execution between operations shortens the overall execution time due to the better utilization of the resources. The amount of overlap depends upon the data dependency between the successive operations. Table 3 shows the execution times for operations following dependent and independent operations.

A vector cross product includes six FPM and three FPS operations, and requires 30 cycles for the SPU to complete these operations. However, if the execution of a vector cross product follows the execution of an independent scalar
addition or multiplication, it may overlap two cycles with the previous scalar instruction; therefore, only 28 cycles are required. The execution time of a vector cross product is further reduced to 24 cycles when it follows the execution of an independent vector addition operation, and the FPM of the SPU is fully utilized under this circumstance. The execution timing charts for the implementations of vector operations using SPU are included in Appendix C.

The overlapped execution times in the rightmost column of Table 2 will be used to estimate the computation times in Figure 49. The computation time for an FPA/FPS operation is either four or zero depending on whether or not it overlaps with a previous FPM operation. Since overlap between FPA and FPM operations occurs about half of the time in this application, two cycles will be used as the estimated FPA execution time.

7.3 Basic Vector Operations Implemented by RVP

This section presents the implementation of basic vector operations with the RVP. The RVP instruction set includes a number of primitive operations such as vector addition, subtraction, multiplication, and vector data transfer operations. The more complex vector operations, such as vector cross product, vector dot product, and matrix-vector multiplication, consist of sequences of these primitive operations.

Table 4 shows the execution times for vector operations following dependent and independent vector operations. The actual implementations for these vector operations are in Appendix D. There are many ways to program the computations. The execution sequence in Appendix D is intended to allow various vector operations to be overlapped without having to modify the detailed execution sequence of each vector operation. For example, the V + V instruction may fully overlap
Table 3: SPU execution time.

<table>
<thead>
<tr>
<th>Previous Instr.</th>
<th>non-overlapped</th>
<th>overlapped with</th>
<th>a + b</th>
<th>a * b, V*V</th>
<th>other v. op's</th>
</tr>
</thead>
<tbody>
<tr>
<td>a + b</td>
<td>4</td>
<td>4</td>
<td>0 %</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>a * b</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>V * V †</td>
<td>14</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>V + V</td>
<td>14</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>V x V</td>
<td>30</td>
<td>28</td>
<td>28</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>V • V</td>
<td>18</td>
<td>16</td>
<td>16</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>MV</td>
<td>42</td>
<td>40</td>
<td>40</td>
<td>36</td>
<td></td>
</tr>
</tbody>
</table>

† V * V is the operation in which each element in the vector is multiplied by the corresponding element in the other vector.

% a + b completely overlaps with a previous a * b operation
with a previous \( V \times V \) operation, therefore, consumes zero RVP instruction cycle. However, this will make the overlapped execution of the following vector operation very difficult.

The execution time for the vector cross product overlapped with a previous vector cross product is 8 cycles. This is three times faster than the computation speed of the RSP. Actually, all vector operations, except the vector dot product, are three times faster than their RSP counterparts. This is the maximum possible speed-up ratio for the parallel execution of \( 3 \times 1 \) vector operations. The poor speed-up performance of the vector dot product is due to the result of the vector dot product is actually a scalar, and its execution can not take advantage of the SIMD structure of the RVP.

The I/O execution time is quite significant for tasks with large I/O requirements. Each vector input occupies an I/O channel for three cycles and then requires a \( WF \) instruction to store the input data from the I/O channel registers, R0 or R1, to the RFs. On the other hand, each vector output requires an OUT instruction to, first, moves the data to the channel registers, and then uses the I/O channel for three more cycles to send the data to the RSP or other RVPs. Since the operations of I/O channels overlap with other RVP operations, the net effect of a vector I/O is that it behaves like a single cycle instruction to the RVP controller and uses only one RVP instruction. The programs to implement the vector operations in Table 4 allow the insertion of two I/O instructions within any four-cycle period without interfering the pipelined execution of the FPMs.

The vector execution time in Table 4 will be used for the remainder of the chapter so that concentration can be placed on parallel implementation at the level beyond vector operations.
Table 4: RVP execution time.

(a) Sequential execution

<table>
<thead>
<tr>
<th>CI</th>
<th>PL</th>
<th>V * V</th>
<th>V + V</th>
<th>V x V</th>
<th>V * V</th>
<th>M V</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>V * V</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>V + V</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>V x V</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>V * V</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>M V</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>20</td>
</tr>
</tbody>
</table>

(b) Overlapped execution with the previous instruction

<table>
<thead>
<tr>
<th>CI</th>
<th>PL</th>
<th>V * V</th>
<th>V + V</th>
<th>V x V</th>
<th>V * V</th>
<th>M V</th>
</tr>
</thead>
<tbody>
<tr>
<td>V * V</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>V + V</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>V x V</td>
<td>12</td>
<td>14</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>V * V</td>
<td>16</td>
<td>17</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>M V</td>
<td>16</td>
<td>17</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

(c) Overlapped execution with the previous two instructions

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>V * V</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>V + V</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>V x V</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>V * V</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>M V</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
7.4 Implementation of Robotic Control With One RSP and One, Two, and Four RVPs

A task graph [9] is a tool to graphically display the composition of computation tasks and the data dependencies between them. Figure 49 is a task graph of the Inverse Plant Plus Jacobian Control algorithm. Each task is represented by a node. The entire control scheme consists of a number of nodes connected by directed arcs. The label in the node identifies the task module, and the directed arcs show the data requirements of the task and the precedence relationships among these nodes.

The numbers inside the node represent estimated execution times. The numbers to the left are estimated execution times using an RVP, and numbers to the right are estimated execution times using an RSP. These estimated execution times are calculated by adding the estimated vector computation time for each vector computation required by the task module. The estimated vector computation times are based on the overlapped RSP and RVP execution times presented in the previous two sections. It should be noted that the data communications delays are not included in the estimate because of their dependencies upon the multiprocessor configuration, task partition, and execution sequence.

Data dependency creates sequential execution as well as communications requirements. As the number of processors increases, so does the communications delay. Eventually, the time saved by the parallel processing of multiple processors is offset by the added communications time. In order to minimize this communication penalty, the task partition should be completed to yield the least amount of off-chip communications requirement.

By totaling all right-hand times in Figure 49, it may be seen that it would take 9762 cycles, plus data transfer delay, to complete the computations with one
Figure 49: Task graph of Inverse Plant Plus Jacobian Control algorithm with estimated execution times.
RSP. In this section, three Synchronized SIMD Array configurations which exploit the parallelism available at the task module level are presented.

The first configuration includes one RSP and one RVP. This configuration allows two tasks to be executed concurrently. Subsection 7.4.1 presents the details of this implementation.

Subsection 7.4.2 presents the implementation using one RSP and two RVPs. Two RVPs will jointly perform computations of task modules to further improve the performance. Since this involves the partitioning of several task modules, with some tightly dependent, it should be expected that speed improvement will be less than ideal.

The computation speed of this system cannot be improved much further by using additional RVPs due to the limited amount of parallelism embedded in the control algorithm. However, it is possible to improve the computation rate of the control system through pipelining which exploits the temporal parallelism. Subsection 7.4.3 presents a one RSP and four RVPs configuration to implement a two-stage pipelined computation.

7.4.1 One RSP and One RVP Implementation

This subsection presents a simple computing structure using one RSP and one RVP. Since two tasks may be executed concurrently on the RSP and RVP, a determination of task partitions and their execution sequence must be made.

Figure 50 is the configuration of the one-RSP-one-RVP implementation. In this figure, the circle represents the RSP, and the square represents the RVP. The four RSP I/O channels are assigned to the upper level command inputs, the plant feedback data, the homogeneous transformation matrices $E$ and $Z$, and the control output $\tau$. After the I/O channels receive input data, they interrupt the SPU, which
then starts the computation. The variables stored in the CM of the RSP include input data, intermediate results of computations, and constant parameters used in the computations. It is assumed that these constant parameters are stored in the CM before the initiation of the computation.

It is very clear from Figure 49 that the scalar task modules, such as LU-decomposition and computation of $\dot{\theta}_d$ and $\dot{\theta}_c$, should be implemented by the RSP because of their scalar computation requirements. Modules such as Force Transform and Outer Loop Control are also good candidates to be implemented by the RSP because of the smaller performance gained by the RVP compared with other modules. The remaining modules are assigned to the RVP because the execution speed of these modules when executed by the RVP is three times faster than by the RSP.

Figure 51 is the task graph for the computation of the Inverse Plant Plus Jacobian Control algorithm using one RSP and one RVP. The total execution time is 3067 cycles for the RVP and 1620 cycles for the RSP. The computation delay for the control scheme is thus 3067 cycles. Figure 52 is the corresponding timing chart of the task graph. The following subsubsections present the implementation of each module by the assigned RSP or RVP.

**Homogeneous Transformation Matrix**

This module is computed by the RVP. The RVP computes the homogeneous matrix elements for three links at the same time: 4, 5, and 6 first, and 1, 2, and 3 after that.

Inputs are originally stored in the CM of the RSP, with the results being returned to the CM. Figure 53 is the task graph for the computations of the Homogeneous Matrices using the RVP. The circles represent the vector computations
Figure 50: Architecture using one RSP and one RVP.
Figure 51: Task graph of Inverse Plant Plus Jacobian Control algorithm with one RSP and one RVP.
Figure 52: Execution timing of the one-RSP-one-RVP implementation.
and the rounded rectangles represent the data communications operations. For the computational nodes, the numbers at the top indicate the sequence of executions, the labels in the middle indicate the operations performed, and the numbers at the bottom represent the computing times required. For the I/O nodes, the upper parts represent the input/output data, the lower parts display the I/O channels used and the actual RVP instruction cycles consumed.

The execution sequence of the vector operations is arranged so that it minimizes the data dependencies between successive operations. The data communications sequence is arranged so that it minimizes the idle time of computational nodes due to unavailable input data.

The Homogeneous Matrices computation takes 350 cycles to complete, from which 301 cycles are used for computation and 49 cycles are used for I/O. Included in the I/O are the input of \( i^{-1}U_i^T \) for \( i = 7, 6, 4, 3 \). This data will be used by the Jacobian module which is to be computed next.

Jacobian

Figure 54 is the task graph for the computation of Jacobian matrix using one RVP. The total execution time is 481 cycles, including 428 cycles of computation and 43 cycles of I/O operations.

Direct Kinematics

Figure 55 is the task graph for the computation of Direct Kinematics using one RVP. The total execution time is 389 cycles, including 372 cycles of computation and 17 cycles of I/O operations. The output homogeneous transformation matrix, \( X \), is sent to the CM and used by the RSP at a later time.
Figure 53: Task graph of Homogeneous matrix computation implemented by one RVP.
Figure 54: Task graph of Jacobian computation implemented by one RVP.
Figure 55: Task graph of Direct Kinematics computation implemented by one RVP.
**Joint Acceleration**

Figure 56 is the task graph for the computation of Joint Acceleration using one RVP. The total execution time is 568 cycles, including 533 cycles of computation and 35 cycles of I/O operations. Nodes 1 through 12 are executed six times before the execution of node 13. The operations in the shaded area are the operations repeated by the following Inverse Dynamics module. Eighteen intermediate results, $\mathbf{1}_i \omega_i, T a_i$, and $T b_i$ are saved to eliminate the redundant computation in the Inverse Dynamics module.

**Inverse Dynamics**

The Inverse Dynamics module includes two parts, the computation of forward recursion and the computation of backward recursion. Figure 57 is the task graph of computations for forward recursion implemented with one RVP. The shaded portion of the task graph is computed during the Joint Acceleration computation. Nodes 1 through 4 are computed six times before the execution of node 5 because they do not depend upon $\hat{\theta}_i$. This allows Inverse Dynamics to overlap with the computation of $\hat{\theta}_i$, which is performed by the RSP.

Figure 58 is the task graph for the computation of backward recursion using one RVP. Nodes 5 through 10 are repeated 6 times. The execution time of the entire Inverse Dynamics module is 1249 cycles.

**Inner Loop Control**

Figure 59 is the task graph for the computation of Inner Loop Control using one RVP. The total execution time is 30 cycles. The results of this module are sent to the CM. If the computation of the Inner Loop Control module is performed by a RSP, the total execution becomes 122 cycles.
Figure 56: Task graph of Joint Acceleration computation implemented by one RVP.
Figure 57: Task graph of Inverse Dynamics forward recursion computation implemented by one RVP.
Figure 58: Task graph of Inverse Dynamics backward recursion computation implemented by one RVP.
Figure 59: Task graph of Inner Loop Control computation implemented by one RVP.
**Force Transform**

Figure 60 is the task graph for the computation of Force Transform using one RSP. The total execution time is 162 cycles including 114 cycles of computation and 48 cycles of I/O operations. The result, $F$, is returned to the CM. The computation of the Force Transform may be performed by the RVP in 122 cycles.

**LU Decomposition of Jacobian Matrix**

Figure 61 is the task graph for the computation of LU decomposition using one RSP. The total execution time is 654 cycles, including 612 cycles of computation and 42 cycles of I/O operations. The input, $J$ matrix, is from the CM, and the outputs, $L$ and $U$ matrices, are returned to the CM. There are six reciprocal computations in the LU decomposition module, with each computation consuming 58 cycles. The method used for the reciprocal computations is the Newton-Raphson iteration method [39]. Appendix E will describe the reciprocal computations in detail. There are many unused instruction cycles during the computation of reciprocals because of the sequential nature of the computation. The execution sequence may be arranged so that the computations of $u_{22}^{-1}, u_{33}^{-1}, u_{44}^{-1}, u_{55}^{-1}$ overlap with the computation of $a_{46}^{(k)}$, and this effectively reduces the actual time consumed by the computations of these reciprocals. Storing the results, $L$ and $U$ matrices, to CM requires only 2 cycles because the storing of each element in these matrices, except $u_{66}^{-1}$, may be performed during the idle cycles.

**Computation of $\tilde{\theta}_d$**

Figure 62 is the task graph for the computation of $\tilde{\theta}_d$ by substitution of the $L$ and $U$ Matrices using one RSP. The total execution time requires 202 cycles,
Figure 60: Task graph of Force Transform computation implemented by one RSP.
Figure 61: Task graph of LU decomposition computation implemented by one RSP.
including 178 cycles of computation and 24 cycles of I/O operations. The output is sent to the CM to be used by the Inverse Dynamics module.

Outer Loop Control

Figure 63 is the task graph for the computation of Outer Loop Control using one RSP. The total execution time is 400 cycles, including 312 cycles of computation and 88 cycles of I/O operations. The Outer Loop Control module may be divided into two parts, OLC A and OLC B. It takes 162 cycles for an RVP to compute OLC A. The OLC B may be computed by an RSP in 214 cycles. The above partition scheme is used to reduce the RSP computation burden in some configurations.

Computation of \( \dot{\theta}_c \)

The computation of \( \dot{\theta}_c \) by substitution of \( L \) and \( U \) Matrices is the same as the computation of \( \ddot{\theta}_d \). The total execution time requires 202 cycles, including 178 cycles of computation and 24 cycles of I/O operations.

7.4.2 One RSP and Two RVPs Implementation

As has been seen, the performance of One-RSP-One-RVP configuration is determined by the computation time of the modules assigned to the RVP. This subsection presents the implementation of the same control scheme using one RSP with two RVPs.

Figure 64 describes the architecture for the control scheme with one RSP and two RVPs. The CHAs of both RVP are connected to CHV of the RSP, and the CHBs are connected on a separate bus. CHAs are used mainly for the communications between the RVPS and the CM. CHBs are used to transfer data between the two RVPs.
Figure 62: Task graph of forward and backward substitution using L and U matrices implemented by one RSP.
Figure 63: Task graph of Outer Loop Control computation implemented by one RSP.
Figure 64: Architecture using one RSP and two RVPs.
Figure 65 is the task graph for the implementation of the control scheme using one RSP and two RVPs. The implementation of the five modules by the RSP remains unchanged. The six task modules originally computed by the single RVP are implemented with the two RVPs according to the following three guidelines:

1. Assign parallel task modules to different RVPs to exploit the parallelism between these modules. For example, the Direct Kinematics and the Joint Acceleration are assigned to different RVPs for parallel execution.

2. Assign tasks requiring the same data to the same RVP to reduce data communication delay. All tasks using $i^{-1}U_i^T$ are assigned to RVP A, and all tasks using $i^{-1}U_i$ are assigned to RVP B.

3. Partition task modules in the critical path to reduce the computation delay of the critical path. The Homogeneous Matrix, Jacobian, and Inverse Dynamics modules are all divided into two subtasks, and are assigned to two RVPs for concurrent executions.

Figure 66 shows the execution timing sequence of the control tasks with the one-RSP-two-RVP configuration. The details of the RVP implementation of these modules are described in the following subsubsections.

**Homogeneous Transformation Matrix**

The computations of homogeneous matrices for each link is completely independent. Instead of computing the homogeneous matrices for three links twice as in the case of one RVP implementation, both RVPs perform the computation for six links concurrently. RVPA computes the homogeneous matrices for links 4, 5, and 6; and RVP-B handles links 1, 2 and 3. The computation sequence is identical
Figure 65: Task graph of one-RSP-two-RVP implementation.
Figure 66: Execution timing of the one-RSP-two-RVP architecture.
to that of the one RVP implementation. The total execution time is 186 cycles. Figure 67 shows the task graph of the Homogeneous Matrix computation.

**Jacobian**

Figure 68 is the task graph of the Jacobian computation by two RVPs. RVPA computes $\frac{7\gamma_i}{\gamma_i}$, the upper half of Jacobian matrix, and RVPB computes $\frac{7\beta_i}{\beta_i}$, the lower half of Jacobian matrix. Since the computations in RVPA requires only $i^{-1}U_i^T$, and the RVPB computations uses only $U_{ij}$, this task partition scheme reduces the register requirements and I/O delays for both RVPs.

RVPA starts computation immediately after it receives the first input vector, at cycle 5; however, RVPB does not start its computations until cycle 23 because of the input delay. RVPA completes the computation of $\frac{7\gamma_i}{\gamma_i}$ by cycle 247. RVPB completes the computation of $\frac{7\beta_i}{\beta_i}$ in 303 cycles. Since the RSP may start executing LU decomposition whenever the upper half of the Jacobian matrix is available, the net effect of the two RVP implementation is that the RSP may start computing Inverse Jacobian after cycle 247 instead of after cycle 481 as was the case of the one RVP implementation.

**Joint Acceleration**

The computation of Joint Acceleration is assigned to RVPA because it requires $U_{ij}^T$ matrices. The computation sequence remains the same as that of the one RVP implementation. The total execution time of the Joint Acceleration is 568 cycles.

**Direct Kinematics**

The computation of Direct Kinematics is assigned to RVPB because the computation uses $U_{ij}$ only. The computation sequence is identical to that of the one
Figure 67: Task graph of Homogeneous matrix computation implemented by two RVPs.
Figure 68: Task graph of Jacobian computation implemented by two RVPs.
RVP implementation. The total execution time of the Direct Kinematics is 389 cycles.

**Inverse Dynamics**

The computation of Inverse Dynamics is partitioned into four subtasks using both RVPs. Figures 69 and 70 are the task graphs of the forward and backward recursions showing the task partitioning between these two RVPs.

Subtask ID.D and ID.C which do not depend on the $\hat{\theta}_d$ are computed by RVPB immediately following the computation of Direct Kinematics. The computations of ID.D consumes 42 cycles and the computations of ID.C uses 244 cycles. These two subtasks are computed first to overlap with the computations of $\theta_d$.

Subtask ID.A starts after the RSP completes the computations of $\hat{\theta}_d$. The computation of the subtask ID.B starts upon the arrival of $i_\omega$ and $i_\bar{q}$. The total execution time for subtasks ID.A and ID.B is 619 cycles.

**7.4.3 Two-Stage Pipelined Implementation with One RSP and Four RVPs**

The amount of parallelism within the Inverse Plant Plus Jacobian Control algorithm has been fully exploited with the one-RSP-two-RVP architecture. The execution time has been reduced from an estimated 9762 cycles using a single RSP to 1938 cycles with the one-RSP-two-RVP architecture. This represents a speed up of about 5 times. Further partitioning of the control algorithm and assigning them to more RVPs do not yield significant speed improvement due to the data dependencies in the algorithm. It is, however, possible to increase the initiation rate of the control system by means of processor pipelining. This subsection presents a pipelined implementation of the control scheme with one RSP and four RVPs.
Figure 69: Task graph of Inverse Dynamics forward recursion computation implemented by two RVPs.
Figure 70: Task graph of Inverse Dynamics backward recursion computation implemented by two RVPs.
Figure 71 is the one-RSP-four-RVP architecture for a two-stage pipelined implementation of the control scheme. CHAs of all four RVPs are connected to the RSP, and the four CHBs are connected by a separate bus. RVPA and RVPB form the first pipeline stage; RVPC and RVPD form the second pipeline stage.

Figure 72 is the task graph for this two-stage pipelined control scheme. The basic task partition is similar to the one-RSP-two-RVP scheme with two differences. First, the Force Transform module is assigned to RVPB. Second, the Outer Loop Control module is split into two subtasks, OLC.A and OLC.B, with OLC.A assigned to RVPB and OLC.B assigned to the RSP. These assignments are intended to reduce the execution time of the RSP because the execution time of the RVP pipeline stage is now shorter than the execution time of the RSP.

The computations of Homogeneous Matrix, Jacobian, Joint Acceleration, Force Transform, and OLC.A are performed during stage one by RVPA and RVPB. The Inverse Dynamics and Inner Loop Control is computed in the second stage by RVPC and RVPD. The $\dot{\theta}_d$ computation, second OLC.B, $\dot{\theta}_c$ computation, and LU decomposition are performed by the RSP. Notice that the LU decomposition is scheduled behind the substitution modules which use the L and U matrices. This means that the first three tasks scheduled for the RSP use the L and U matrices computed from the previous LU decomposition. In other words, the computation of LU decomposition may be considered as the first pipeline stage, and the computations of the other three modules may be considered as the second pipeline stage.

Figure 73 is the timing chart of the computation of the control scheme. The length of the pipeline stage is determined by the computation time of the RSP, 1272 cycles. Since the RSP is also responsible for the supervision of activities such
Figure 71: Architecture using one RSP and four RVPs.
Figure 72: Task graph of Inverse Plant Plus Jacobian Control algorithm using one RSP and four RVPS.
as external I/O handling and tasks scheduling, it is reasonable to expect the actual pipeline stage to be slightly longer than 1272 cycles. Assuming the other activities consume 28 SPU cycles, the pipeline latency is 1300 cycles. This represents an increase of initiation rate of 49 percent. The execution delay of the pipelined architecture is around 2207 cycles, which is 269 cycles longer than the execution time of the one-RSP-two-RVP implementation. Whether the 49 percent increase in initiation rate with 14 percent increase in execution time is desirable requires further studies.

### 7.5 Multirate Implementation With Multiple RSPs and RVPs

The previous section has exploited the parallelism at the task module level by partitioning the task modules and then implementing them with multiple RVPs. The data dependency embedded in these modules has limited the performance improvement as the number of RVPs used in the computing structures increased. This subsection presents three more configurations using multiple RSPs to further improve control performance by exploiting the multirate characteristics of a real-time control system.

Figure 74 is the same task graph as Figure 49 plus four feedback loops. The paths from the commanded input to the control output are not critical because the commanded inputs change at a much slower rate. The four paths shown in the figure are: 1) the inner loop of Jacobian control, 2) the outer loop of Jacobian control, 3) the Inverse Plant loop, and 4) the Inverse Jacobian loop. These four loops have different impacts to the control output $\tau$. Some loops should be executed more often than others in order to obtain better control performance. Tsai [47] and Lilly [3] have studied the multirate control scheme by applying controls at different rates for different control paths. It is found from their studies that 10:3:1:1 is a
Figure 73: Execution timing of the one-RSP-four-RVP architecture.
reasonable ratio of execution rate between these four loops. The inner loop requires
the highest execution rate of 10, the outer loop is next with 3, and the remaining
two loops execute at rate of 1.

Subsection 7.5.2 presents a dual-rate configuration with two RSPs and two
RVPs. One RSP is dedicated to the computation of Inner Loop Control path, and
the remaining three loops are implemented by the other RSP and its associated
two RVPs.

Subsection 7.5.3 presents a triple-rate implementation of three RSPs and four
RVPs. Subsection 7.5.4 presents a quadruple-rate implementation with four RSPs
and seven RVPs.

7.5.1 Dual-Rate Implementation Using Two RSPs and Two RVPs

Figure 75 is the architecture of the dual-rate control implementation with
two RSPs and two RVPs. The two RSPs are connected via their I/O channels.
Figure 76 is the tasks graph of the dual-rate implementation. The execution of
Inner Loop Control task at the bottom of the figure is performed by a dedicated
RSP, RSPA. All the other tasks are implemented by RSPB and its associated two
RVPs. The task assignment and sequencing is the same as the one-RSP-two-RVP
implementation except that the Inner Loop Control module is computed separately
by the other RSP.

Figure 77 is the timing chart of the dual-rate implementation. The inner loop
requires 122 cycles of execution time. The I/O interrupt will occupy some of the
processor cycles. The interrupt routine for this node is very simple and requires
only a few instructions to set the I/O channels status. Assuming the I/O interrupt
handling uses 28 cycles per iteration, the gross execution time for the inner loop
is 150 cycles.
Figure 74: Task graph of Inverse Plant Plus Jacobian Control algorithm showing four feedback loops and estimated execution time.
Figure 75: Architecture for dualrate implementation using two RSPs and two RVPs.
Figure 76: Task graph of Inverse Plant Plus Jacobian Control algorithm using the two-RSP-two-RVP architecture.
Figure 77: Execution timing of the dualrate implementation of the two-RSP-two-RVP architecture.
The remaining modules are executed every 1908 cycles by the other RSP and its two RVPs. The inner loop computation is executed about 12 times more frequent than the other loops.

7.5.2 Triple-Rate Implementation Using Three RSPs and Four RVPs

The dual-rate control scheme presented in the previous subsection increased the computation rate of the inner loop. This subsection presents a triple-rate control scheme which computes the outer loop control at a fast rate.

Figure 78 shows the architecture of the triple-rate control implementation with three RSPs and four RVPs. All three RSPs are interconnected. The task scheduling is shown in Figure 79. The inner loop control is performed by RSPA. The nodes in the outer loop are assigned to RSPB and its associated two RVPs. The Inverse Plant and Jacobian loop are assigned to RSPC and another two RVPs connected to it.

Figure 80 is the timing chart for the triple-rate control implementation. The inner loop execution time is approximated to be 150 cycles. The execution rate of the outer loop control is determined by the computation time of RSPB, 604 cycles.

The Jacobian and Inverse Plant loop are implemented by RSPC and its two RVPs. The task sequencing is similar to the original two-RVP implementation without the tasks not involved in these two loops. The execution time is still 1908 cycles because the tasks removed are not in the critical path of the original implementation. The ratio of execution rate among these four loops is now around 12:3:1:1.

7.5.3 Quadruple-Rate Implementation Using Four RSPs and Seven RVPs

The triple-rate architecture has produced the desired relative execution for the outer loop control. This section is to further decouple the execution of Jacobian
Figure 78: Architecture for triplerate implementation using three RSPs and four RVPs.
Figure 79: Task graph of Inverse Plant Plus Jacobian Control algorithm using the three-RSP-four-RVP architecture.
Figure 80: Execution timing of the triplerate implementation of the three-RSP-four-RVP architecture.
and Inverse Plant loops in order to obtain a quadruple-rate control scheme in which each loop is computed independently.

Figure 81 is the architecture for the quadruple-rate implementation with four RSPs and seven RVPs. RSPs are connected according to the data communication requirements. For example, RSPA is connected to RSPB to receive $\tau_d$ and RSPB is connected to RSPC to receive $L$ and $U$ matrices from RSPC. The Inverse Plant loop is implemented by a RSP and four RVPs as a two-stage pipeline.

Figure 82 is the task graph for this control scheme. The execution timing is shown in Figure 83. The inner loop and outer loop control are implemented the same way as in the previous section. The Jacobian loop is performed by RSPC and RVPC. The execution rate of the Jacobian loop is determined by the computation delay of the computation of the Homogeneous Matrix and Jacobian Matrix which require 831 cycles.

The data dependency of Inverse Plant loop encourages the pipeline implementation. A two-stage pipeline scheme is employed here to divide the execution time in half. The execution sequence is similar to the two-stage pipeline introduced early. The pipeline latency is 754 cycles.

The quadruple-rate implementation has reduced the execution time of the Inverse Plant and Jacobian loops to 754 and 831 cycles respectively. This makes the relative computation rate among the four loops be 10:2.5:2:1.

7.6 Summary

In this chapter, several different configurations built upon the RSP and RVP to implement the Inverse Plant Plus Jacobian algorithm have been studied. This section will evaluate the performance of these various configurations.
Figure 81: Architecture for quadruplerate implementation using four RSPs and seven RVPs.
Figure 82. Task graph of Inverse Plant Plus Jacobian Control algorithm implementation using the four-RSP seven-RVP architecture.
Figure 83: Execution timing of the quadruplerate implementation of the four-RSP-seven-RVP architecture.
Table 5(a) summarizes the computation time for each of the configurations presented in this chapter plus two estimated execution times for a RSP and a processor without any form of parallelism. The RVP processor cycle is used as the unit in this table. Table 5(b) shows the computation time in microseconds for the estimated 125 ns cycle time. Since test results of the FPA chip show that the FPA functions properly at 80 ns cycle time, Table 5(c) presents the computation time for an 80 ns cycle time.

In Table 1, the total number of floating point additions, floating point multiplications, and integer operations were given. Using the floating point adder and multiplier designed in chapter 6, a single-bus, register oriented processor without any parallel and pipelined execution would require 9 cycles to execute an FPM instruction, 7 cycles to execute an FPA instruction, 5 cycles to execute an integer instruction. This total number of cycles required to complete the computation of the control algorithm is 30823. This estimation does not consider the time required for memory and I/O communications.

The execution time using one RSP is also estimated based on the results from Sections 7.2 and 7.3. This is the implementation using the first layer of the four-layer architecture presented in Chapter 3. It takes an estimated 9762 cycles for the RSP to complete the computation of this algorithm. This estimate does not consider memory and I/O time either. This represents a speed-up of 3.15 over the previous case. The speed-up lies mainly in the overlapped instruction fetch, the simultaneous dual-bus operands fetch, the temporary FPA and FPM output registers, the pipelined FPM execution, and the overlapped FPA and FPM execution.
Table 5: Performance comparison between various configurations.

a) Execution times and pipeline latencies based on RVP cycles

<table>
<thead>
<tr>
<th></th>
<th>Synchronized Implementation</th>
<th>Multirate Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1-bus processor with no parallelism</td>
<td></td>
</tr>
<tr>
<td>Entire Control Algorithm</td>
<td>1 RSP</td>
<td>1 RVP</td>
</tr>
<tr>
<td>E T</td>
<td>30823</td>
<td>9782</td>
</tr>
<tr>
<td>Inner Loop</td>
<td>P L</td>
<td>-</td>
</tr>
<tr>
<td>Outer Loop</td>
<td>P L</td>
<td>-</td>
</tr>
<tr>
<td>Inverse Plant Loop</td>
<td>P L</td>
<td>-</td>
</tr>
<tr>
<td>Jacobian Loop</td>
<td>P L</td>
<td>-</td>
</tr>
</tbody>
</table>

b) Execution times and pipeline latencies based on 125 ns RVP cycle time

<table>
<thead>
<tr>
<th></th>
<th>Synchronized Implementation</th>
<th>Multirate Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1-bus processor with no parallelism</td>
<td></td>
</tr>
<tr>
<td>Entire Control Algorithm</td>
<td>1 RSP</td>
<td>1 RVP</td>
</tr>
<tr>
<td>E T</td>
<td>3652.8</td>
<td>1220.2</td>
</tr>
<tr>
<td>Inner Loop</td>
<td>P L</td>
<td>-</td>
</tr>
<tr>
<td>Outer Loop</td>
<td>P L</td>
<td>-</td>
</tr>
<tr>
<td>Inverse Plant Loop</td>
<td>P L</td>
<td>-</td>
</tr>
<tr>
<td>Jacobian Loop</td>
<td>P L</td>
<td>-</td>
</tr>
</tbody>
</table>

c) Execution times and pipeline latencies based on 80 ns RVP cycle time

<table>
<thead>
<tr>
<th></th>
<th>Synchronized Implementation</th>
<th>Multirate Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1-bus processor with no parallelism</td>
<td></td>
</tr>
<tr>
<td>Entire Control Algorithm</td>
<td>1 RSP</td>
<td>1 RVP</td>
</tr>
<tr>
<td>E T</td>
<td>2465.8</td>
<td>780.9</td>
</tr>
<tr>
<td>Inner Loop</td>
<td>P L</td>
<td>-</td>
</tr>
<tr>
<td>Outer Loop</td>
<td>P L</td>
<td>-</td>
</tr>
<tr>
<td>Inverse Plant Loop</td>
<td>P L</td>
<td>-</td>
</tr>
<tr>
<td>Jacobian Loop</td>
<td>P L</td>
<td>-</td>
</tr>
</tbody>
</table>

211
The next layer in the architecture considers the parallelism embedded in the basic $3 \times 1$ vector operations. The configuration using a RSP and a RVP is able to perform the control tasks in 3067 cycles. This is a speed-up of 3.18 over the single RSP implementation, or a speed-up of 10.0 over the non-parallel single-bus processor. The speed-up comes from the parallel computation between the RSP and RVP, the simultaneous computation among the three FPPs, the fast inter-FPP data transfer using the on-chip dual Shift Networks, and the overlapped I/O and CPU operations with the dual I/O channels.

The third layer, the synchronized SIMD array, exploits the parallelism embedded in the computation of the control algorithm. A configuration using one RSP and two RVPs has been presented to speed up the computation by partitioning the control tasks. The execution time is reduced to 1938 cycles. This a speed-up of 1.58 over the one-RSP-one-RVP configuration, or a speed-up of 15.9 over the single-bus processor. The concurrent execution of tasks is the key to the improved performance.

In the three cases discussed above, parallel and pipelined execution are geared toward speeding up the computation of the control task. It is found that the majority of the parallelism in the application has been exploited by the one-RSP-two-RVP configuration. The speed improvement produced by further task partitioning with additional RVPs is limited by the data dependency between tasks. The remaining configurations are intended to improve the control performance through increasing the computation initiation rate.

A two-stage pipeline scheme is implemented by the one-RSP-four-RVP configuration. Sequential tasks assigned to the vector processors are performed in two stages. The pipeline latency of the control system is reduced to 1300 cycles. This represents an improved initiation rate of 1.49 over the one-RSP-two-RVP
implementation. The actual execution time is increased from 1938 to 2207 cycles.

These one RSP implementations discussed above perform the control at single rates which do not take advantage of the multirate characteristics of the real-time control environment. The next three configurations use two, three, and four RSPs to implement the multirate control schemes. These configurations belong to layer 4 of the hierarchical architecture.

The dual-rate configuration using two RSPs and two RVPs reduces the pipeline latency of the Inner Loop to 150 cycles while the remaining loops are executed every 1008 cycles. The triple-rate implementation using three RSPs and four RVPs further reduces the Outer Loop latency to 604 cycles. Finally, the Quadruple-rate implementation using four RSPs and seven RVPs allows each loop to be controlled by its own RSP at a different rate. The Inverse Plant Loop latency is reduced to 831 cycles, and the Jacobian Loop latency is reduced to 754 cycles. These time delays are 37 to 50 times shorter than the computation time of the non-parallel single-bus processor.

These implementations have demonstrated the effectiveness of the layered VLSI architecture in dealing with the various parallelism embedded in the real-time robotic control algorithm.
CHAPTER VIII
SUMMARY AND CONCLUSIONS

8.1 Research Contributions

The objective of this dissertation research has been to design a layered restructurable multiprocessor computer architecture using state-of-the-art VLSI technology to perform the control of robotic systems in real-time. The research includes three major areas: the architectural design for the parallel processing of robotic computation, the mapping of the Inverse Plant Plus Jacobian Control algorithm onto the designed architecture, and the design and layout of the VLSI processors specified in the architecture.

The main contributions of this dissertation research include:

1. The design of a restructurable multiprocessor architecture based upon two VLSI processors to exploit the various parallelism embedded in the real-time robotic control applications in a layered fashion

2. The microarchitecture design of the Robotic Vector Processor which achieves a maximum 18 degrees of parallelism for the efficient computation of 3x1 vector operations.

3. The microarchitecture design of a Robotic Scalar Processor which complements the computing power of the RVP with circuits to perform scalar
computation, RVP sequencing, and I/O communications in a highly overlapped fashion.

4. The CMOS design, layout, simulation, and testing of a 32-bit floating point adder which was tested to execute a floating point addition in 320 ns.

5. The CMOS design, layout, simulation, and testing of a 32-bit floating point multiplier.

6. The CMOS design, layout, and simulation of a compact dual-port register cell for the register file.

7. The CMOS design, layout, simulation, and testing of I/O pads for the scalable 1.2 to 3.0 μm design rules.

8. The implementation of the Inverse Plant Plus Jacobian Control algorithm using various computing structures built upon the two VLSI processors.

The architectural design in this research has followed a systematic approach by evaluating the concurrency and data dependency embedded in the various levels of computation. The result is a hierarchical multiprocessor architecture consisting of four layers: the scalar, the SIMD, the synchronized SIMD array, and the decentralized MIMD. Each layer in the computing hierarchy is responsible for its own targeted concurrency, and serves as the underlining structure for the next layer.

The layered approach has transformed the task of parallel architecture design into the definition of layers, each with its own targeted parallelism, and interfaces between them. This approach has decoupled the considerations of several different architectural issues and addressed each of them separately at the appropriate layers.
Two VLSI processors, the Robotic Scalar Processor (RSP), and the Robotic Vector Processor (RVP), has been identified as the building blocks for the layered architecture. These two processors are used repeatedly to construct various computing structures for different computational tasks. This approach has two major advantages. First, only two types of VLSI chips are required. Second, these two processors are able to configure optimal structures for the computational tasks.

The architecture of the RVP has been designed. The RVP includes three floating point processors, two interconnection networks, and two I/O channels. This processor is designed specifically for the computation of 3x1 vector operations used extensively in the robotic computation. The floating point processor consists of a 32-bit floating point adder, a floating point multiplier, and a large register file. Its architecture is designed to fully exploit low level parallelism via pipelining, overlapped execution, and bus multiplexing. By simplifying the controller and removing the control sequencer from the RVP, all three floating point processors and the interconnection networks may be implemented on the same chip. The on-chip interconnection networks plus the overlapped executions of data transfers and floating point operations have effectively eliminated the communication delays between these floating point processors.

The RVP has been designed keeping in mind the fact that several RVPs be configured to jointly perform well-defined computation tasks such as the Inverse Dynamics module of the Inverse Plant Plus Jacobian Control algorithm.

The RSP has been designed to handle the scalar computations, the sequencing of vector operations, the data communications for the associated RVP structure, and the global communications. The RSP complements the computational speed of the RVP by performing all the data communications, and the scheduling and synchronization of parallel processes within the multiprocessor system.
The RVP chip consists of six major functional blocks: the floating point adder, the floating point multiplier, the register file, the shift and broadcast network, the I/O channel, and the processor controller. The design of the RVP was carried out to the actual layout of the floating point adder, floating point multiplier, and dual-port register file. CAD tools from UC Berkeley were used to layout and simulate these circuits.

Both the floating point adder and the floating multiplier were sent to MOSIS for fabrication in 3.0 μm CMOS technology. The floating point adder was fabricated and 13 samples were returned within three months. The floating point test chips were tested by a Tektronix 9100 DAS test station. Four samples were tested, and all functioned correctly. The floating point adder was tested to function at an 80 ns clock period, 45 ns faster than the simulated speed. If the floating point multiplier, which is due to arrive soon, can operate at the same rate, the RVP has a computing power of 18.75 MFLOPS.

Several 3.0 μm I/O pads for the scalable CMOS design rules were designed in house due to the lack of suitable cells in the cell library consistent with the imposed design rules. The layout style employed in the RVP design has carefully taken into account of the various characteristics of CMOS circuit. The template used for the data path design allowed the circuits to be laid-out in a very straightforward manner. This has reduced the layout design time significantly. The layout of the floating point adder test chip and floating point multiplier test chip, containing 6762 and 11121 transistors respectively, required about six months to complete using the SUN 3/160C workstation.

The layered architecture is used to implement the Inverse Plant Plus Jacobian Control algorithm. The control task was first implemented using a pair of RSP and RVP, which required 3067 cycles. This represents a speed-up of about 3.18
over the implementation using a single RSP. An implementation with one RSP and two RVPs were then studied and the computation has been reduced to 1938 cycles. The amount of parallelism embedded in the algorithm has been advantageously exploited using the two-RVP configuration. The next configuration uses an RSP and five RVPs to implement a two stage processor pipeline to increase the computation rate. This configuration performs the computation of the control task every 1300 cycles.

Configurations with multiple RSPs have been studied to exploit the multirate characteristics of the real-time control system. First, a dual-rate structure with two RSPs and two RVPs has been implemented with one of the RSP dedicated to the critical control loop, the inner loop. A triple-rate configuration with three RSPs and four RVPs has also been studied. At last, a quadruple-rate configuration using four RSPs and seven RVPs has been implemented, allowing the inner loop to be executed every 150 cycles, and the other loops to be executed every 604, 754 and 831 cycles. For a simulated 125 ns cycle time, this represents a control implementation of 53,000 Hz for the inner loop and 9600 to 13,000 Hz for the other loops.

8.2 Research Extensions

This dissertation has designed a layered restucturable multiprocessor architecture for the real-time control of robotic applications. Even though the Inverse Plant Plus Jacobian Control algorithm was used as the implementation example, the architecture is able to handle a wide range of real-time control applications. It is hoped that the layered implementation methodology for the multiprocessor architecture design will lead to better multiprocessor architecture designs. The following is a brief description of possible future research.
1. **RVP.** The architectural design and VLSI test results have shown significant computation power attainable from the RVP chip. The majority of the design is completed and tested. It would be a significant step to complete the remaining design and fabricate the RVP under 1.25 $\mu m$ technology. Since there is enough silicon area to implement a large size of the register file on the RVP, a register file of 128 words would increase the data reference locality significantly. The number of I/O channels could be increased to four to increase the configuration flexibility. An on-chip matrix transposition circuit may be useful to reduce the dependency on the Common Memory.

2. **RSP.** The RVP is useless if not complemented by the RSP. Its implementation is as important as the RVP. The Common Memory is the key of the communications between the vector, scalar, and I/O operations. Further research on the design of the memory to reduce the access time would be significant. The global communication scheme may be refined to improve the flexibility of Common Memory utilization. In addition, research may continue in the direction of implementing global memory scheme through the RSP memory bus. This would complement the message-passing scheme of the RSP I/O channels.

3. **Software Tools.** The programming method used in this dissertation is quite primitive. To ease the programming of RSPs and RVPs, it is necessary to have a proper compiler and simulation program to relieve the user from manually synchronizing the operations at various processors.

4. **Scheduling Algorithm.** The tasks partitioning and scheduling used in this dissertation are ad hoc approaches. More systematic methods for the task
partitioning and scheduling would be quite useful.

5. **Other Robotic Applications.** The architecture of the RSP and RVP are not limited to the implementation of the Inverse Plant Plus Jacobian Control algorithm. Applications such as the Inverse Kinematics may also be implemented with this architecture.

6. **Heterogeneous SIMD Array Structure.** By designing different types of vector processors, such as processor optimized for 4x1 or other types operations, it is possible to configure a heterogeneous SIMD array structure for a wide variety of applications.

7. **Cube Architecture.** The RSP may be configured as a cube structure. Each node, the RSP, has an associated vector computing structure optimized for different types of computations. This structure would be able to perform more general types of computational applications.

8. **Software Layers.** Just like the hardware, the software design should be layered so that applications can be implemented logically with separate layers.

9. **Higher Precision.** For certain applications, such as the Inverse Kinematics, the precision of the 32-bit floating point format is not sufficient. A 40-bit floating point format with 31-bit mantissa may prove to be worthy of the extra silicon area and delay.

10. **Fault Tolerance.** Another major consideration for the architecture design for real-time control is the ability to function under certain fault conditions. Fault tolerant design may be applied at the arithmetic unit level or be applied at the processor level. Software implementation should also be considered.
REFERENCES


[34] Sherburne, R.W., Processor Design Tradeoffs in VLSI, Ph.D. dissertation, University of California, Berkeley, April, 1984.


[45] Terman, C., ESIM: a switch level simulation program, MIT.


223


This appendix presents the computational tasks of the Inverse Plant Plus Jacobian Control algorithm developed by Orin [1]. Lilly, in her thesis [3], has described the control scheme in full detail. Most of the equations used in this appendix are directly borrowed from her thesis. The reader may refer to [3] for further information.

For the purpose of demonstration of the multiprocessor implementation, this chapter will deal with the case of six-link robot manipulator plus an end effector. The base of the manipulator may move with respect to the fixed reference frame. All links are connected by revolute joints. The remainder of Appendix A will describe the equations for each module used in the control scheme.

**Homogeneous Matrix**

The Homogeneous Matrix module computes the homogeneous transformation matrices. Upon receipt of input data $\theta_i$, the joint angle for link $i$, the homogeneous transformation matrices need to be recalculated. The homogeneous transformation matrix, $^{i-1}T_i$, may be computed as follows:
The matrix \( i^{-1}T_i \) may be separated into a 3 x 3 orientation transformation matrix, \( i^{-1}U_i \), and a 3 x 1 translational transformation vector, \( i^{-1}p_i^* \), as follows:

\[
i^{-1}T_i = \begin{bmatrix}
\cos \theta_i & -\sin \theta_i \cos \alpha_i & \sin \theta_i \sin \alpha_i & a_i \cos \theta_i \\
\sin \theta_i & \cos \theta_i \cos \alpha_i & -\cos \theta_i \sin \alpha_i & a_i \sin \theta_i \\
0 & \sin \alpha_i & \cos \alpha_i & d_i \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

The sin \( \theta_i \) and cos \( \theta_i \) may be approximated as follows:

\[
\sin \theta_i = \theta_i - \frac{1}{3!} \theta_i^3 + \frac{1}{5!} \theta_i^5 - \frac{1}{7!} \theta_i^7 + \frac{1}{9!} \theta_i^9 - \frac{1}{11!} \theta_i^{11} + \frac{1}{13!} \theta_i^{13}
\]

\[
\cos \theta_i = 1 - \frac{1}{2!} \theta_i^2 + \frac{1}{4!} \theta_i^4 - \frac{1}{6!} \theta_i^6 + \frac{1}{8!} \theta_i^8 - \frac{1}{10!} \theta_i^{10} + \frac{1}{12!} \theta_i^{12} - \frac{1}{14!} \theta_i^{14}
\]

**Direct Kinematics**

The Direct Kinematics module calculates the actual position of the end effector, \( X \). For a manipulator with 6 degrees of freedom, \( X \) is computed as follows:

\[
X = Z \cdot T_1 \cdot T_2 \cdot T_3 \cdot T_4 \cdot T_5 \cdot T_6 \cdot E
\]

Each one of the matrices in the above equation, including \( Z \) and \( E \), is a 4 x 4 homogeneous transformation matrix. The multiplication of two 4 x 4 homogeneous
matrices may be decomposed into operations of $3 \times 3$ matrices and $3 \times 1$ vectors as follows:

$$i^{-1}T_i \cdot i^i T_{i+1} = \begin{bmatrix} i^{-1}U_i & i^{-1}E_i^* \\ O_{1\times 3} & 1 \\ i^{-1}U_i & i^{-1}E_i + i^{-1}P_i^* \\ O_{1\times 3} & 1 \end{bmatrix}$$

**Force Transform**

The force transformation module converts the $6 \times 1$ vector of forces and moments applied by the end effector, $\mathbf{wF}$, from wrist coordinates to end effector coordinates. The computation is as follows:

$$\mathbf{F} = \begin{bmatrix} f \\ n \end{bmatrix} = \begin{bmatrix} i^{-1}U_i & O_{3\times 3} \\ (e_{P_w} \times) e_u & e_u \end{bmatrix} \begin{bmatrix} \mathbf{wF} \\ \mathbf{wN} \end{bmatrix}$$

**Jacobian**

The Jacobian matrix translates the end effector velocity, $\dot{\mathbf{x}}$, to the joint velocity vector, $\dot{\mathbf{\theta}}$, as follows:

$$\dot{\mathbf{x}} = J(\mathbf{\theta}) \dot{\mathbf{\theta}}$$

where $J(\mathbf{\theta})$ is the Jacobian matrix. The Jacobian matrix is computed as follows:
\[\tau U_7 = I\]
\[\tau U_{i-1} = \tau U_i^{-1} U_{i-1}^T\quad i = 7,\ldots,2,1\]
\[\tau \bar{X}_i = \tau U_{i-1} [0\ 0\ 1]^T\quad i = 6,\ldots,2,1\]
\[\tau L_7 = 0\]
\[\tau r_{i-1} = \tau r_i - \tau U_i \tau p_i^*\quad i = 7,\ldots,2,1\]
\[\tau \beta_i = \tau \gamma_i \times (-\tau r_{i-1})\quad i = 6,\ldots,2,1\]
\[J = \begin{bmatrix} \gamma_1 & \gamma_2 & \gamma_3 & \gamma_4 & \gamma_5 & \gamma_6 \\ \beta_1 & \beta_2 & \beta_3 & \beta_4 & \beta_5 & \beta_6 \end{bmatrix}\]

**Inverse Jacobian**

The inverse Jacobian module converts \(\dot{x}_c\) to \(\dot{\theta}_c\) as follows:

\[\dot{\theta}_c = J^{-1}\dot{x}_c\]

The Inverse Jacobian computation involves two steps:

1. Perform the LU decomposition for the \(J\) matrix, \(J = [j_{ij}] = LU = [u_{ij}][k_{ij}]\).

2. Perform forward and backward substitution to solve for \(\dot{\theta}_c\).

The LU decomposition [9] without pivoting is described as follows:

\[j_{ij}^1 = j_{ij}\quad i = 1,\ldots,5,6; j = 1,\ldots,5,6\]
\[j_{ij}^{k+1} = j_{ij}^k + l_{ik}(-u_{kj})\quad i = 1,\ldots,5,6; j = 1,\ldots,5,6\]
\[l_{ik} = \begin{cases} 0 & \text{if } i < k \\ 1 & \text{if } i = k \\ j_{ik}^k u_{kk}^{-1} & \text{if } i > k \end{cases}\]
\[u_{kj} = \begin{cases} j_{kj}^k & \text{if } k > j \\ 0 & \text{otherwise} \end{cases}\]
Joint Acceleration

The Joint Acceleration module converts $\ddot{x}_d$ to $\ddot{\theta}_d$. The computation is as follows:

\begin{align*}
0_{l0} & = 0_{\omega_0} \\
i_{l_i} & = i^{-1}U_i^T(i^{-1}l_{i-1} + i^{-1}\omega_{i-1} \times [0 0 \dot{\theta}_i]^T) & i = 1,\ldots,6 \\
\gamma_{l7} & = \gamma U_6(0_{l6}) \\
0_{b0} & = 0_{\rho_0} \\
i_{h_i} & = i^{-1}U_i^T(i^{-1}h_{i-1} + i l_i \times i\rho_i^* + i\omega_i \times (i\omega_i \times i\rho_i^*)) & i = 1,\ldots,6 \\
\gamma_{h7} & = \gamma U_6(0_{h6}) \\
i_{\eta_i} & = \begin{bmatrix} i_{l_i} \\
i_{h_i} \end{bmatrix} \\
\ddot{\theta}_d & = J^{-1}(\ddot{x}_d - \gamma_{h7})
\end{align*}

Inverse Dynamics

The Inverse Dynamics is to compute the necessary actuator forces/torques, $\tau_d$, from $\theta, \dot{\theta}, \ddot{\theta}_d$, and $F_d$.

The computation consists of two parts, the forward recursion and the backward recursion. The equations for the forward recursion are:

\begin{align*}
i_{\omega_i} & = i^{-1}U_i^T(i^{-1}\omega_{i-1} + [0 0 \dot{\theta}_i]^T) & i = 1,\ldots,6 \\
i_{\dot{\omega}_i} & = i^{-1}U_i^T(i^{-1}\dot{\omega}_{i-1} + [0 0 \ddot{\theta}_i]^T + i^{-1}\omega_{i-1} \times [0 0 \dot{\theta}_i]^T) & i = 1,\ldots,6 \\
i_{\rho_i} & = i^{-1}U_i^T(i^{-1}\rho_{i-1}) + i\omega_i \times i\rho_i^* + i\omega_i \times (i\omega_i \times i\rho_i^*) & i = 1,\ldots,6 \\
i_{\ddot{\rho}_i} & = i\omega_i \times i\ddot{\rho}_i + i\omega_i \times (i\omega_i \times i\ddot{\rho}_i) + i\rho_i & i = 1,\ldots,6 \\
i_{E_i} & = m_i i_{\ddot{\rho}_i} & i = 1,\ldots,6 \\
i_{N_i} & = iH_i(i\omega_i) + i\omega_i \times (iH_i i\omega_i) & i = 1,\ldots,6
\end{align*}
The backward recursion is computed as follows:

\[
\begin{align*}
6f_7 &= -6U_7[O_{3\times 3} | I_{3\times 3}](-E_d) \\
6\eta_7 &= 6p_i^* \times 6f_7 - 6U_7[I_{3\times 3} | O_{3\times 3}](E_d) \\
^{i-1}f_i &= i-1U_i(iE_i + i f_{i+1}) \\
^{i-1}\eta_i &= i-1U_i\{i\eta_{i+1} + i E_i + i p^*_i \times i f_{i+1}\} \\
\tau_{di} &= i-1n_i^2
\end{align*}
\]

**Outer Loop Control**

The Outer Loop Control module implements the force and position feedback to calculate \( \dot{x}_c \). The equations are:

\[
\begin{align*}
\dot{x}_c &= \dot{x}_d + k_1(X_d \odot X) + k_2(E_d + F) \\
&= \dot{x}_d + k_1D_{PE} + k_2(E_d + F)
\end{align*}
\]

where

\[
\begin{align*}
X_d &= \begin{bmatrix} n_d & a_d & a_d & p_d \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
X &= \begin{bmatrix} n & a & a & p \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
\frac{1}{2}\{(a \cdot a_d) - (a \cdot a_d)\} \\
\frac{1}{2}\{(n \cdot a_d) - (a \cdot n_d)\} \\
D_{PE} &= \begin{bmatrix} n \cdot (p_d) - p \\ a \cdot (p_d) - p \end{bmatrix}
\end{align*}
\]

230
Inner Loop Control

The Inner Loop Control module implements joint rate feedback to calculate the torques to be applied to the joint actuator, \( \tau \). The computation is:

\[
\tau = \tau_d + k_3(\dot{\theta}_c - \dot{\theta})
\]
APPENDIX B

CIRCUIT DESIGN OF THE ROBOTIC VECTOR PROCESSOR

B.1 Floating Point Adder

B.1.1 Controller

Figure 84 is the circuit diagram of the FPA controller. The input signals are \( \text{fpa}, \phi_1, \) and \( \phi_2 \) from the RVP controller. The controller is basically a five-stage shift register consisting of five quasi-static CMOS master-slave latches using \( \phi_1 \) and \( \phi_2 \) as input clocks. The outputs are buffered with large inverters to provide greater driving current. Figure 85 shows the controller output waveforms.

B.1.2 Input Latch

The inputs to the latches from BA and BB are latched by the master-slave latches. Figure 86 is the transistor logic diagram of a CMOS master-slave latch. The latch is a quasi-static circuit, therefore, may be stopped in the inactive state without loss of data stored by the latch. The master-slave latch requires a setup time of 14\,\text{ns}. The output delays range from 8 to 15\,\text{ns} for load from 50 \,\text{fF} to 1000 \,\text{fF}.

B.1.3 Operand Comparator

The 31-bit operand, both exponent and mantissa, comparator is implemented by a Manchester carry propagation chain with partial carry lookahead. The function of the comparator may be described by

232
a) Implementation of FPA controller with five master-slave latches.

![Diagram of FPA controller with master-slave latches]

b) Detailed circuit of a CMOS quasi-static master-slave latch.

![Diagram of CMOS quasi-static master-slave latch]

Figure 84: FPA controller
Figure 85: FPA controller output waveforms

Figure 86: FPA input latch.
\[ AgtB_i = p_i \cdot AgtB_{i-1} + g_i; \quad i = 1, \ldots, 31 \]

where

\[ p_i = a_i \cdot b_i + a_i \cdot \bar{b}_i, \]

\[ g_i = a_i \cdot \bar{b}_i, \]

and

\[ AgtB_0 = 0. \]

Figure 87 shows the block diagram of the comparator. The 31-bit carry chain is divided into one 3-bit ripple carry and seven 4-bit carry lookahead segments. Subsection 6.5.1 has described the design of the Manchester carry chain circuit in full detail.

The carry output, \( AgtB_{\text{out}} \), is used to control the switching of the exponents and mantissas. The switching of the exponent can take place as soon as the comparator finishes the comparison of the exponents. The exponent switching becomes true whenever one of the eight exponent bits generates a carry out of the comparator.

**B.1.4 Operand Switch**

The switching circuit is implemented with pass transistors and is followed with output inverters. Figure 88 is the transistor logic diagram of one of the 32 switches. Pass transistor logic is used here due to its simplicity and spatial efficiency. The output inverter restores the output signal level and prevents any back feed from the load.

**B.1.5 Mantissa Alignment Shift Controller**

Following the switch circuit, the difference between the exponents is calculated to provide the control signals to the alignment circuit which includes 5 fixed-step shifters of step sizes 1, 2, 4, 8, and 16 bits.
Figure 87: FPA 31-bit Manchester carry propagation comparator with 4-bit carry lookahead blocks.
Figure 88 is the circuit diagram of the shift controller which includes the exponent subtractor. This 8-bit subtractor is implemented with two 4-bit dynamic Manchester carry chains. The propagate and generate signals are derived according to the following:

\[ p_i = a_i \odot b_i, \text{ and} \]
\[ g_i = a_i \cdot \bar{b}_i; \text{ for } i = 0, ..., 7. \]

The input carry of the least significant bit is set to 1, and the carry ripples through pass transistors along the carry chain. The difference is calculated by

\[ d_i = \bar{c}_i \odot p_i; \text{ for } i = 0, ..., 7. \]

The resulting difference ranges from 0 to 255. The three most significant bits of the difference are ORed to provide an OVR24 signal which is then ORed with
bits 3 and 4 of the subtractor. Bits 0 through 4 are then used to control the five alignment shifters.

B.1.6 Mantissa Alignment Circuit

The mantissa alignment circuit is 24 bits wide. The circuit right shifts the selected mantissa from 0 to 24 bits depending on the difference of the exponents. The alignment circuit includes a total of 120 shifter cells. Figure 90 is the circuit diagram of a single shifter cell. Notice that the output is buffered. The worst case propagation delay through each shifter is $9\text{ns}$. 

Figure 89: FPA mantissa alignment shifter controller.
B.1.7 Sign and Mantissa CPA/CPS Operation Generator

The actual mantissa adder/subtractor operation is derived from the input operator Sub-Add, $S_a$, and $S_b$ as

$$cpa-op = S_a \oplus S_b \oplus Sub-Add$$

Sign bit of the FPA output is calculated by exclusively NORing $cpa-op$ with the sign bit of the greater operand, $S_{max}$.

B.1.8 Mantissa Carry Propagation Adder/Subtractor

The 24-bit mantissa adder/subtractor is implemented by six 4-bit Manchester carry propagation adder/subtractors. Figure 91 is the block diagram of the mantissa carry propagation adder/subtractor.

Figure 90: FPA mantissa alignment shifter cell.
Figure 91: FPA 24-bit Manchester carry propagation adder/subtractor
Figure 92(a) shows the detailed circuit for \( p \) and \( g \) generation, and Figure 92(b) shows the circuit for \( \text{sum/difference} \) generation. The function of these blocks are described as follows

\[
\begin{align*}
bb_i &= b_i \odot \text{Sub-Add} \\
c_0 &= \text{Sub-Add} \\
p_i &= a_i \oplus bb_i, \\
g_i &= a_i \cdot bb_i, \\
\text{sum}_i &= p_i \odot \overline{c_i}, \\
\overline{c_{i+1}} &= \overline{c_i} \cdot p_i + g_i; \quad \text{for } i = 0, \ldots, 23.
\end{align*}
\]
a) Circuit for $p$ and $g$ generation.

b) Circuit for Sum/difference generation.

Figure 92: Circuits for carry propagation adder/subtractor.
B.1.9 Leading Zeros Remover

The leading zeros remover includes five leading zeros detectors and five left shifters of steps 16, 8, 4, 2, and 1. The shifting circuits are similar to the mantissa alignment circuit except that the shifters for the leading zero remover shift to the left. Figure 93 is the circuit diagram of an 8-bit leading zero detector. This is a pseudo-nmos circuit with a pull-up transistor which is on all the time.
B.1.10 Overflow and Underflow Handler

Overflow flow occurs when the resultant biased exponent exceeds 255. Underflow occurs when the resultant biased exponent is negative or when the resultant mantissa is zero. The overflow (Ovrfl) and underflow (Undfl) signals are fed to the clamping circuit which forces all exponent and mantissa bits to 1 when Ovrfl is true and forces all bits to 0 when Undfl is true. Figure 94 is the logic diagram of the overflow/underflow handler. The two NOR gates at the bottom of the figure form one of the 31 clamping cells.
B.1.11 Output Latch

Figure 95 is the transistor logic diagram of the output latch. The output is connected to the processor busses BA and BB. The inverter $Q_{out}$ in front of the output pass transistor is included to isolate the internal data from being altered by the output.
B.2 Floating Point Multiplier

B.2.1 FPM Controller

The control signal generation circuit is shown in Figure 96. This is also a shift register like that of the FPA. The assertion of the fpm signal starts the generation circuit. The resulting output waveforms are shown in Figure 97.
a) Implementation of FPM controller using 6 master-slave latches.

b) Detailed circuit of a master-slave latch.

Figure 96: FPM controller.
Figure 97: FPM controller output waveforms.
B.2.2 Mantissa Shifter

The input latches of the FPM are the same as the FPA latches. The mantissa B input latch is modified to implement a shifting function as shown in Figure 98. When fpm is high, the operands from BB are loaded. When fpm is low, mantissa B is shifted up by six bits. Each cycle, six least significant bits are shifted to the CSA array multiplier to form six rows of partial products.
Ain = Abar nor Bbar
Sout = Ain xor Cin xor Sin
Cout = Ain Cin + Ain Sin + Cin Sin

Figure 99: FPM carry save adder cell.

B.2.3 Carry Save Adder Multiplier

Figure 99 is the logic diagram of the carry save adder cell. \( C_{in} \) is first exclusively ORed with \( A_{in} \), and the result then exclusively ORed with \( S_{in} \) to produce \( S_{out} \). This arrangement forces the delay paths to alternate between sums and carries, and thus reduces the overall multiplier delay. If \( S_{in} \) instead of \( C_{in} \) is first exclusively ORed with \( A_{in} \), the total delay of the six array CSA multiplier will be 235.8 ns which is 28.8 ns longer than this design. The CSA array performs a
24-bit mantissa multiplication in 4 cycles. During first cycle, the CSA multiplier $S_{in}$ and $C_{in}$ selectors force them to zeros. After the first cycle, $S_{in}$ and $C_{in}$ are feedback from the CSA pipeline register. Figure 100 shows the circuit of a CSA multiplier $S_{in}$ and $C_{in}$ selector. After the fourth cycle, the result of the CSA array is latched for the following carry propagation adder while the CSA array performs the computation for the next FPM operation. Figure 101 shows the CSA multiplier pipeline register and the CPA input latch.

**B.2.4 31-bit Exponent and Mantissa Carry Propagation Adder**

Figure 102 is the block diagram of the 31-bit exponent and mantissa carry propagation adder. Whenever the mantissa adder generates a carry output, the exponent need to be incremented by one. The cascading of the exponent and mantissa adders handles this situation by feeding the carry output of the mantissa adder to the exponent adder as the input carry. Carry lookahead blocks of length
Figure 101: FPM CSA multiplier pipeline register and CPA input latch.

4 and 8 are used to reduce the carry propagation delay.

B.3 Shift and Broadcast Network Shift Register

Figure 103 is the circuit diagram of a shift register of the SBNs. The register may shift data forward, backward, or not at all. Broadcasting is performed by turning on the R4/R5 pass transistor.

B.4 I/O Channel Shift Register

Figure 104 is the circuit diagram of a shift register of the I/O channels. The register shifts data in one direction.
Figure 102: FPM 31-bit exponent and mantissa carry propagation adder.
Figure 103: FPM Shift and Broadcast Network shift register.
Figure 104: I/O channel shift register.
B.5 Input Pad

Figure 105 is the circuit diagram of the input pads designed for the FPA and FPM test chips. A resistor implemented using a polysilicon area (320λ x 16λ) is used to absorb part of the energy of input spikes. The voltage at node A is protected by two large diodes which clamp the voltage to within 0 to 5 volts. A large n-transistor $T_A$ with a permanently grounded gate is also used to protect the internal circuits by discharging the input signal to substrate in the event of overvoltage. The two inverters are used to improve the signal waveform and to provide the internal circuits with two complementary signals. The input pad has an 8 ns delay for an internal load of 400 fF.
Figure 106: Output pad.

B.6 Output Pad

Figure 106 is the circuit diagram of the output pads designed for the FPA and FPM test chips. The output signal is amplified by four cascading transistors of increasing driving capacities. The outputs of the third and fourth transistors are then used to control the two n-transistors of the last stage. The reason for using an n-transistor for the pull-up is to take advantage of better mobility of the electrons. The output is a TTL compatible signal. The delay of an output pad driving eight input pads requires 17 ns for pull-up and 12 ns for pull-down.
APPENDIX C

ROBOTIC SCALAR PROCESSOR IMPLEMENTATION OF BASIC VECTOR OPERATIONS

Appendix C presents the detailed implementations of vector computations using the Scalar Processor Unit of Robotic Scalar Processor. The implementations include:

1. Program and execution timing for vector cross product.

2. Program and execution timing for vector dot product.

3. Program and execution timing for matrix vector multiplication.

Figures 107 through 109 show the details of these vector operations using one RSP.
\[
X \times Y = Z = \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} \times \begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \end{bmatrix} = \begin{bmatrix} X_2 \cdot Y_3 - X_3 \cdot Y_2 \\ X_3 \cdot Y_1 - X_1 \cdot Y_3 \\ X_1 \cdot Y_2 - X_2 \cdot Y_1 \end{bmatrix}
\]

a) Program

1. \text{fpm 0, } X_2, Y_3
2. \text{nop}
3. \text{fpm 0, } X_3, Y_2
4. \text{nop}
5. \text{fpm 0, } X_3, Y_1
6. \text{fps 0, } R_3, R_3
7. \text{fpm 0, } X_1, Y_3
8. \text{nop}
9. \text{fpm 0, } X_1, Y_2
10. \text{fps } Z_1, R_3, R_3
11. \text{fpm 0, } X_2, Y_1
12. \text{nop}
13. \text{nop}
14. \text{fps } Z_2, R_3, R_3
15. \text{nop} \quad \% \text{Instruction 16 is not necessary}
16. \text{fps } Z_3, R_0, R_0 \quad \% \text{if the following fps will write the result back to Register File.}

b) Timing chart

Figure 107: RSP implementation of vector cross product.

259
\[
X \cdot Y = Z = \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} \cdot \begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \end{bmatrix} = X_1 \cdot Y_1 + X_2 \cdot Y_2 + X_3 \cdot Y_3
\]

a) Program

1. fpm 0, X1, Y1
2. nop
3. fpm 0, X2, Y2
4. nop
5. fpm 0, X3, Y3
6. fpa 0, R3, R3
7. nop
8. fpa 0, R2, R3
9. nop
10. fps Z, R0, R0 % Instruction 10 is not necessary if the following fpa/fps will write the result back to Register File.

b) Timing chart

![Timing Chart]

computation time = 9 cycles

Figure 108: RSP implementation of vector dot product.
\[
MV = [M1 \ M2 \ M3] \ [V] = Z
\]
\[
= \begin{bmatrix}
m_{11} & m_{12} & m_{13} \\
m_{21} & m_{22} & m_{23} \\
m_{31} & m_{32} & m_{33}
\end{bmatrix}
\begin{bmatrix}
v_1 \\
v_2 \\
v_3
\end{bmatrix}
= \begin{bmatrix}
m_{11} * v_1 + m_{12} * v_2 + m_{13} * v_3 \\
m_{21} * v_1 + m_{22} * v_2 + m_{23} * v_3 \\
m_{31} * v_1 + m_{32} * v_2 + m_{33} * v_3
\end{bmatrix}
\]

a) Program
1. \text{fpm 0, } m_{11}, v_1
2. \text{nop}
3. \text{fpm 0, } m_{12}, v_2
4. \text{nop}
5. \text{fpm 0, } m_{13}, v_3
6. \text{fpa 0, } R_{3d}, R_3
7. \text{fpm 0, } m_{21}, v_1
8. \text{fpa 0, } R_2, R_3
9. \text{fpm 0, } m_{22}, v_2
10. \text{nop}
11. \text{fpm 0, } m_{23}, v_3
12. \text{fpa 0, } R_{3d}, R_3
13. \text{fpm 0, } m_{31}, v_1
14. \text{fpa 0, } R_2, R_3
15. \text{fpm 0, } m_{32}, v_2
16. \text{nop}
17. \text{fpm 0, } m_{33}, v_3
18. \text{fpa } Z_2, R_{3d}, R_3
19. \text{nop}
20. \text{fpa 0, } R_2, R_3
21. \text{fpa 0, } R_2, R_3
22. \text{fpa } Z_3, R_0, R_0, \%

% Instruction 22 is not necessary if the following fpa/fps will write the result back to Register File.

b) Timing chart

![Timing chart]

computation time = 21 cycles

Figure 109: RSP implementation of matrix-vector multiplication.
APPENDIX D

RVP IMPLEMENTATION OF BASIC VECTOR OPERATIONS

Appendix D presents the detailed implementations of vector computations using the Robotic Vector Processor. The implementations include:

1. Program, execution timing, and reservation table for vector cross product.
2. Program, execution timing, and reservation table for vector dot product.
3. Program, execution timing, and reservation table for matrix vector multiplication.
4. Overlapped vector operations with a previous $V \times V$ operation.
5. Overlapped vector operations with a previous $V + V$ operation.
6. Overlapped vector operations with a previous $V \times V$ operation.
7. Overlapped vector operations with a previous $MV$ or $V \cdot V$ operation.

Figures 110 through 116 show the detailed implementations of these vector operations using one RVP.
\[ X \times Y = Z = \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} \times \begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \end{bmatrix} = \begin{bmatrix} X_2 \cdot Y_3 - X_3 \cdot Y_2 \\ X_3 \cdot Y_1 - X_1 \cdot Y_3 \\ X_1 \cdot Y_2 - X_2 \cdot Y_1 \end{bmatrix} \]

a) Program

1. lgr X, Y
2. sh bwd, fwd
3. fpm R2, R3
4. nop
5. nop
6. sh bwd, fwd
7. fpm R2, R3
8. nop
9. nop
10. nop
11. nop
12. nop
13. fpa R7d, R7
14. nop
15. nop
16. nop
17. wfe R6, Z

b) Timing chart

```
1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21

17 cycles
```

c) Reservation Table for FPP1

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>fpm</td>
<td>a</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fpm</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fpm</td>
<td>a</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R7</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R7d</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fpa</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fpa</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 110: Program, execution timing, and reservation table for vector cross product.

263
\[ X \cdot Y = Z = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} x_1 \cdot y_1 + x_2 \cdot y_2 + x_3 \cdot y_3 \\ x_1 \cdot y_1 + x_2 \cdot y_2 + x_3 \cdot y_3 \\ x_1 \cdot y_1 + x_2 \cdot y_2 + x_3 \cdot y_3 \end{bmatrix} \]

a) Program

\begin{itemize}
  \item 1. \text{lcr X, Y}
  \item 2. \text{fpm R2, R3}
  \item 3. \text{nop}
  \item 4. \text{nop}
  \item 5. \text{sh bwd, bwd}
  \item 6. \text{fpm R2, R2}
  \item 7. \text{sh bwd, bwd}
  \item 8. \text{nop}
  \item 9. \text{nop}
  \item 10. \text{fpm R2, R3}
  \item 11. \text{nop}
  \item 12. \text{fpa R7d, R7}
  \item 13. \text{nop}
  \item 14. \text{nop}
  \item 15. \text{nop}
  \item 16. \text{fpa R6, R7}
  \item 17. \text{nop}
  \item 18. \text{nop}
  \item 19. \text{nop}
  \item 20. \text{wf R6, Z}
\end{itemize}

b) Timing chart

\begin{itemize}
  \item 1 3 5 7 9 11 13 15 17 19 21
  \item lcr \rightarrow sh
  \item + 20 cycles
  \item sh +
  \item wf |
\end{itemize}

c) Reservation Table for FPP1

\begin{align*}
a &= x_1 \cdot y_1; & b &= x_2 \cdot y_2; & c &= x_3 \cdot y_3; & d &= a + b; & z &= d + c
\end{align*}

\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
& a & b & c & d & e & f & g & h & i & j & k & l \\
\hline
lcm & a & a & a & a & b & b & b & b & b & b & b & b & b \\
\hline
fpm 1 & a & a & a & b & b & b & b & b & b & b & b & b & b \\
\hline
fpm 2 & a & a & a & a & a & a & a & a & a & a & a & a & a \\
\hline
R7 & a & a & a & a & b & b & b & b & b & b & b & b & b \\
\hline
R7d & a & a & a & a & b & b & b & b & b & b & b & b & b \\
\hline
fpa & d & d & d & d & z & z & z & z & z & z & z & z & z \\
\hline
\end{tabular}

Figure 111: Program, execution timing, and reservation table for vector dot product.
\[
MV = [M1 \ M2 \ M3] \ V = Z
\]

\[
= \begin{bmatrix}
m11 & m12 & m13 \\
m21 & m22 & m23 \\
m31 & m32 & m33 \\
\end{bmatrix}
\begin{bmatrix}
v1 \\
v2 \\
v3 \\
\end{bmatrix}
= \begin{bmatrix}
m11 \cdot v1 + m12 \cdot v2 + m13 \cdot v3 \\
m21 \cdot v1 + m22 \cdot v2 + m23 \cdot v3 \\
m31 \cdot v1 + m32 \cdot v2 + m33 \cdot v3 \\
\end{bmatrix}
\]

a) Program

1. lsr V, R3
2. fpm R4, M1
3. nop
4. nopfpm R4, M3

b) Timing chart

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
<th>11</th>
<th>13</th>
<th>15</th>
<th>17</th>
<th>19</th>
<th>21</th>
</tr>
</thead>
</table>
| lsr | V, R3 | fpm | R4, M1 | nop | fpm | R4, M3 | nop | fpm | R4, M1 |...

<table>
<thead>
<tr>
<th>12</th>
<th>14</th>
<th>16</th>
<th>18</th>
<th>20</th>
</tr>
</thead>
</table>
| lsr | V, R3 | fpm | R4, M1 |...


20 cycles

---

<table>
<thead>
<tr>
<th>22</th>
<th>24</th>
<th>26</th>
<th>28</th>
<th>30</th>
<th>32</th>
<th>34</th>
<th>36</th>
<th>38</th>
<th>40</th>
<th>42</th>
</tr>
</thead>
</table>
| 20 cycles |...

---

c) Reservation Table for FPP1

\[a = m11 \cdot v1; \ b = m12 \cdot v2; \ c = m13 \cdot v3; \ d = a + b; \ z = d + c\]

\begin{array}{cccccccccccccccc}
\hline
fpm \_1 & & a & a & a & b & b & b & c & c & & & & & & & & & & & & & \\
R7 & & & & & & a & a & a & b & b & b & c & c & c & & & & & & & & & \\
\end{array}

Figure 112: Program, execution timing, and reservation table for matrix vector multiplication.
Figure 113: Timing of overlapped vector operations with a previous V * V operation.
Figure 114: Timing of overlapped vector operations with a previous V + V operation.
Figure 115: Timing of overlapped vector operations with a previous $V \times V$ operation.
Figure 116: Timing of overlapped vector operations with a previous MV or $V \cdot V$ operation.
APPENDIX E

RECPROCAL GENERATION USING RSP

The technique for calculating reciprocals is based on the Newton-Raphson method for obtaining the roots of the equation

\[ F(X) = \frac{1}{X} - B. \]

The roots of the equation can be found by iteratively evaluating the equation

\[ X_{i+1} = X_i - F(X_i)/F'(X_i) \]
\[ = X_i - \frac{1}{X_i} - B)/(X_i^2) \]
\[ = X_i(2 - B \times X_i). \]

The process begins by making a guess as to the value of \( X_0 \), and using this guess or "seed" value to perform the first iteration. Iterations are continued until the root is evaluated to the desired accuracy. The number of iterations needed to achieve a given accuracy depends on the accuracy of the seed value. The error of \( X_i \) reduces quadratically. The number of bits of accuracy in the result, then, roughly doubles after every iteration. For a seed of 4-bit accuracy, it requires three iterations to obtain an accuracy of over 24 bits. The following is a five-step SPU program to generate a seed of 4-bit accuracy.

1. \( F_B = B \) and 100000000011110000000000000000000. This extracts the sign and first four bits of mantissa.

270
2. $\bar{F}_X = F_B \text{ xor } 00000000011110000000000000000000$. This complements the first four bits of the mantissa.

3. $E_B = B \text{ and } 01111111000000000000000000000000$. This extracts the exponent of B.

4. $E_{X_0} = 01111110100000000000000000000000 \text{ sub } E_B$. This calculates the exponent of the reciprocal($E_{X_0} = 253 - E_B$)

5. $X_0 = E_{X_0}$ or $\bar{F}_{X_0}$. 