A DATA-DRIVEN MULTIPROCESSOR ARCHITECTURE FOR HIGH THROUGHPUT DIGITAL SIGNAL PROCESSING

by

Kwok Wah Yeung

Memorandum No. UCB/ERL M95/54

10 July 1995

A DATA-DRIVEN MULTIPROCESSOR ARCHITECTURE FOR HIGH THROUGHPUT DIGITAL SIGNAL PROCESSING

by

Kwok Wah Yeung

Memorandum No. UCB/ERL M95/54

10 July 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

A Data-driven Multiprocessor Architecture for High Throughput Digital Signal Processing

by

Kwok Wah Yeung

Doctor of Philosophy in Engineering -
Electrical Engineering and Computer Sciences

University of California at Berkeley

Professor Jan M. Rabaey, Chair

A data-driven multiprocessor architecture called **PADDI-2** specially designed for rapid prototyping of high throughput digital signal processing algorithms is presented. Characteristics of typical high speed DSP systems were examined and the efficiencies and deficiencies of existing traditional architectures were studied to establish the architectural requirements, and to guide the architectural design. The proposed PADDI-2 architecture is a highly scalable and modular, multiple-instruction stream multiple-data stream (MIMD) architecture. It consists of a large number of fine-grain processing elements called *nanoprocessors* interconnected by a flexible and high-bandwidth communication network. The basic idea is that a data flow graph representing a DSP algorithm is directly mapped onto a network of nanoprocessors. The algorithm is executed by the nanoprocessors executing the operations associated with the assigned data flow nodes in a data-driven manner. High computation power is achieved by using multiple nanoprocessors to exploit the large amount of fine-grain parallelism inherent in the target algorithms. Programming flexibility is provided by the MIMD control strategy and the flexible interconnection network which can be reconfigured to handle a wide range of DSP algorithms, including those with heterogeneous communication patterns.

As a proof of concept, a single-chip multiprocessor integrated circuit containing 48 16-bit nanoprocessors was designed and fabricated in a 2-metal 1-μm CMOS technology. A 2-level, area-efficient communication network occupying only 17% of the core area provides flexible and high-bandwidth inter-processor communications. Running at 50 MHz, the chip achieves 2.4 GOPS peak performance and 800 MBytes per second I/O bandwidth. An integrated development system including an assembler, a VHDL-base system simulator, and a demonstration board has been developed for PADDI-2 program development and demonstration.

The benchmark results based on a variety of DSP algorithms in video processing, digital communication, digital filtering, and speech recognition confirm the performance, efficiency and generality of the architecture. Moreover, when compared with several competitive architectures that target the same application domain, PADDI-2 is in general about 2 to 3 times better in terms of hardware efficiency.

[illegible]

Jan M. Rabaey
Committee Chairman

To my loving family

Table of Contents

List of Figures

List of Tables

Acknowledgments

First, I would like to thank my advisor, Professor Jan Rabaey. The PADDI-2 project would not be possible without his vision, leadership, support, and patience. His vision and guidance was instrumental in the definition of this project and in every stage of its development. He is truly a great teacher and an exceptional researcher.

Professors John Wawrzynek, David Aldous, and Bernhard Boser all provided early guidance for my research as members of my Qualifying Exam Committee. Professors Wawrzynek and Aldous went above and beyond the call of duty by also serving on my dissertation committee and reviewing this thesis. Professor Wayrzynek's comments and suggestions have greatly improved the presentation of the benchmark results in this thesis.

I would like to thank a number of my coworkers who have contributed to the PADDI-2 project. Besides being a supportive office-mate, Arthur Abnous did a superb job in developing the assembler and the VHDL model of the PADDI-2 system. Nelson Chow and Stas Frumkin helped tremendously in debugging the PADDI-2 chip and in mapping the first couple of algorithms onto the chip. My sincere thanks to Matt Power who endured the tedious task of designing the test board on Racal. Stephen Okelo-odongo simulated the controller functions on the test board. Dev Chen, who designed the first-generation PADDI architecture, gave me valuable advice on research and life. I would also like to thank the other members of Jan's research group - Ole Bentz, Lisa Guerra, Sean Huang, Paul Landman, My Le, Seungjun Lee, David Lidsky, Renu Mehra, Miodrag Potkonjak, Roy Sutton, Ingrid Verbauwhede, and Engling Yeo - for providing an enjoyable research environment.

I would also like to express my gratitude to Sam Sheng, Kevin Zimmerman, Sue Mellers, and Brian Richards for supporting the computers, laboratory equipments, and CAD tools. On the administrative side, I would like to thank Tom Boot, Peggy Brown, and Corey Schafer.

My thanks also go out to Tom Burd, Andy Burstein, Anantha Chandrakasan, Phu Hoang, Timothy Hu, Gani Jusuf, Timothy Kam, Wook Koh, Monte Mar, Shankar Narayanaswamy, Eric Ng, Mani Srivastava, Lars Thon, Jane Sun, Greg Uehara, Scarlett Wu, and Robert Yu. They made the past couple of years in Cory so much more bearable. I specially would like to thank my coworker Cormac Conroy. Besides solving many of my technical problems, he has also corrected the zillions of grammatical errors in the draft of this thesis.

Lastly, I would like to recognize the members of my family. My sisters Winnie, Mindy and Cheryl have encouraged me every step of the way. To my parents, I owe more to you than I could ever repay. Thank you for your unconditional love and support.

And to my fiancee, Marlisa. Thank you for your love and warmth that keep me going through both good and bad time.

CHAPTER 1

Introduction

Leave the beaten track occasionally, and dive into the woods. You will be certain to find something that you have never seen before.

Alexander Graham Bell

1.1 Perspective

Because of the numerous advantages in processing signals in digital form as opposed to analog form, e.g., accuracy, stability and repeatability, digital signal processing (DSP) has become and will remain a dominant force in signal processing and communications [73]. Impacts can be seen in practically every field, for example, consumer and home electronics, digital communications, medical, manufacturing, and defense. Typical applications are digital audio [5, 68], high definition digital television [53, 87], digital image and video processing [42], computer graphics [79, 81], speech recognition and synthesis [7, 82], mobile communications [44, 70], personal communications systems [15], robotics and electro-mechanical control, machine vision [11], sonar and radar [80], smart weapons, and target discrimination and tracking [65].

After identifying a new market, the designers of modern DSP systems are faced with two key problems: optimizing the performance and cost of the system, and shortening the design cycle. The latter can be equally if not more important than the former. Because of the inertia of customers against adopting new products and the continual shortening of product life cycles, short time-to-market is critical in order to capture market share of any products.

A design process typically goes through many iterations of the design-simulate-evaluate cycle. Traditional simulation methodology is based on software modelling of the system at various levels of abstraction. However, as the customer's demands on higher performance and DSP algorithms become more and more complex, software-based simulation can be the bottleneck in the design process.

Even with rapid advances in computer technology and high-level system modelling, software simulation cannot meet the throughput requirement of complex systems, especially real-time systems. For example, to simulate the high-definition color digital camera [60] on a 10-second video script requires a throughput on the order of GOPS. Furthermore, if the simulation is too slow for real-time playback, Giga-bytes of memories are needed to store the video for later playback. Very often, subjective and perceptual measures, the so-called "golden-eyes and golden-ears", as opposed to deterministic measures, e.g., signal-to-noise ratio and mean-square-error, are used to evaluate the quality of a design. Many hours of video scripts and tapes must be listened to and watched throughout the design cycle. In these cases, a real-time prototype is especially needed to enhance the ability of the designers to fine-tune many design parameters and to explore new design space.

In addition, it is not an easy task to model the behavior of the external world with which the system interacts. For example, to test out a new communication protocol, the channel characteristics under various traffic loads have to be modelled accurately. A hardware prototype operating in real-time can be hooked up to the actual physical channel and can provide the designers accurate performance metrics under realistic operating conditions. In many cases, hardware prototypes are constructed solely to verify theoretical results with experimental data [95].

Although a real-time hardware prototype is clearly an invaluable tool for shortening the design cycle of complex DSP systems, previous approaches in hardware prototyping are not satisfactory and suffer from one or more of the following shortcomings: high development costs, insufficient computation power, slow turnaround, susceptibility to errors, difficulty in testing and debugging, inflexibility, limited scalability, restrictive programmability, and lack of compiler support. (More of these approaches will be discussed in Chapter 3.) The problem is aggravated when handling complex DSP algorithms with high sampling rates and heterogeneous data-flow patterns. This motivates us to research a high throughput programmable multiprocessor architecture called PADDI-2 that overcomes many of the aforementioned shortcomings and can handle rapid prototyping of these types of algorithms.

1.2 Goals and Contributions

In this chapter, we have presented the needs for rapid prototyping in hardware that form the motivation for this research. The main goal of our research is to study and define an architectural template of a flexible field-programmable integrated circuit component that can be applied to rapid prototyping of a wide range of real-time high speed DSP systems. In addition to architectural level design, another goal is to realize the proposed architecture in silicon to study the important implementation-related issues which can have major impact on architectural choices.

The major contribution of this research is the design and implementation of a novel multiprocessor architecture called PADDI-2 [113, 114]. Characteristics of typical high speed DSP systems were examined to establish the architectural requirements. Multiprocessor architectures were classified based on the granularity of processing elements. The analysis of the classification results lead to the investigation of multiprocessor architectures using fine-grain processing elements. This class of architectures provides a nice balance between flexibility and high computation power, both of which are essential for rapid prototyping of high throughput DSP algorithms.

The proposed PADDI-2 architecture is a multiple-instruction stream multiple-data stream (MIMD) multiprocessor architecture. It consists of a large number of simple processing elements called nanoprocessors interconnected by a flexible and high-bandwidth communication network. The basic idea is that a data flow graph representing a DSP algorithm is directly mapped onto a network of nanoprocessors. The algorithm is executed by the nanoprocessors executing the operations associated with the assigned data flow nodes in a data-driven manner. High computation power is achieved by using multiple nanoprocessors to exploit the large amount of fine-grain parallelism inherent in many target algorithms. Programming flexibility is provided by the MIMD control strategy and the flexible interconnection network which can be reconfigured to handle a wide range of DSP algorithms, including those with heterogeneous communication patterns.

As a proof of concept, a single-chip multiprocessor integrated circuit was designed and fabricated in a 2-metal 1-μm CMOS technology [116]. It consists of 48 nanoprocessors organized in 12 clusters of 4 each. A 2-level, area-efficient communication network occupying only 17% of the core area provides flexible and high-bandwidth inter-processor communications. In addition to the direct interconnections between neighboring nanoprocessors, a total of 7.2 GBytes per second communication bandwidth is available in the Level-1 network for communications within the 12 clusters, and up to 4.8 GBytes per second bandwidth for inter-cluster communications can be supported by the Level-2 network. Hardware mechanisms handle data-driven operations and inter-processor synchronization. Running at 50 MHz, the chip achieves 2.4 GOPS peak performance and 800 MBytes per second I/O bandwidth. The chip was tested functional on second silicon and simple algorithms have been demonstrated. An integrated software environment which includes an assembler and a VHDL-base system simulator has been developed to facilitate program development and debugging on PADDI-2. An SBus-based board containing eight PADDI-2 chips is being designed for demonstration of more complex DSP systems.

Although the concepts of fine-grain processing elements and MIMD multiprocessing have been around for a long time, we believe the idea of applying both concepts in our application domain is original. Existing architectures typically apply single-instruction stream multiple-data stream (SIMD) style of control for fine-grain processing elements and MIMD for coarser grain processing elements. By targeting only high throughput DSP algorithms with plenty of fine-grain parallelism, we argued that a local program store of small capacity (less than 8 instructions in our prototype chip) is sufficient and therefore the control overhead associated with the MIMD approach can be significantly reduced. In addition, thanks to rapid advances in VLSI process technology, device density has improved so much that area overhead due to MIMD control and elaborate interconnect network can be tolerated to a much larger extent.

PADDI-2 is an experimental architecture developed to prove that it makes sense to use the MIMD approach even for very fine-grain processing elements. The highly scalable MIMD architecture, a clean programming paradigm based on the data flow concept and a flexible interconnection network not only enhance programmability but also improve the computation throughput by making it easier to utilize the large processing power provided by the fine-grain processing elements.

Indeed, benchmark results based on a variety of DSP algorithms in image and video processing, digital communications, digital filtering, and computer graphics verify the performance and generality of the PADDI-2 architecture. Moreover, when compared with several competitive architectures that also target the same application domain, PADDI-2 is in general about 2 to 3 times better in terms of hardware efficiency. The benchmark results will be presented in Section 6.7.

1.3 Overview

The following is a chapter by chapter outline of the remainder of this thesis.

Chapter 2 High Speed Digital Signal Processing

Chapter 2 introduces the class of real-time high speed DSP algorithms targeted by this research. Several typical applications in this class, including a Viterbi Detector for Disk-drives, a video engine for hidden surface removal for 3-D computer graphics, and a high definition digital color camera system, are examined. Their requirements on computation power, functional operations, data flow patterns, I/O and memory bandwidth, control functions, granularity, and amount of parallelism are discussed and summarized.

Chapter 3 Rapid Prototyping Platforms

Efficiencies, limitations and trade-offs of existing hardware prototyping approaches are first discussed. It is then argued that approaches based on programmable, multiprocessing hardware offer the most cost-efficient solution to satisfy the performance and flexibility requirements. Programmable multiprocessor IC architectures for high throughput DSP are analyzed and classified according to the granularity of the basic processing elements and the control/arithmetic ratio. Representative architectures are cited and compared. Special attention is given to the architectures that exploit parallelism in DSP algorithms using very fine-grain processing elements.

Chapter 4 PADDI2: Architectural Design

The PADDI-2 architecture is presented in Chapter 4. First, the architectural requirements for rapid prototyping of DSP algorithms are summarized. Next, the control structure, processing element architecture, communication network design, memory and I/O architecture of the PADDI-2 architecture are described, and key design choices are explained. Various techniques used to achieve high performance are discussed, and alternative architectures are suggested and discussed.

Chapter 5 PADDI-2: Hardware Design

In Chapter 5, the VLSI design of the key components of the PADDI-2 chip, the processing elements, the communication network and the scan unit, are presented. Important circuit simulation results and testing/debugging strategies are also described.

Chapter 6 PADDI-2: Development System

The programmer model of PADDI-2 is first reviewed in Chapter 6. We then describe the development system of PADDI-2, which includes an assembler for low-level programming, a VHDL-base system simulator for simulation and debugging support, and a VME board for hardware demonstration and scan-testing. The architecture of a multi-PADDI-2 board currently being designed is also covered. We next present examples of the mapping of practical DSP algorithms onto the PADDI-2 architecture. The chapter is concluded by reporting and analyzing benchmark results of practical DSP applications, and comparing the results with several competitive architectures.

Chapter 7 Conclusions and Future Work

Chapter 7 concludes the dissertation. Alternative PADDI-style architectures are discussed and future research directions are mentioned.

CHAPTER 2

High Speed Digital Signal Processing

In this chapter, several real-time, high speed DSP applications and their implementations in the fields of communications, computer graphics, and video processing are used to illustrate the application domain targeted by this research. Characteristics of the algorithms such as computation power, functional operations, data flow patterns, I/O and memory bandwidth, control functions, granularity and amount of parallelism are discussed. Typical features of the target algorithms are summarized. These applications will be referred to when architectures are evaluated and compared in the next chapter.

A common measure of the communication bandwidth requirement of an algorithm is the I/O bandwidth of the whole algorithm. A more realistic and also meaningful measure of the communication bandwidth requirement is the worst case I/O bandwidth of the whole system as well as all the sub-systems. For instance, if the algorithm is to implemented by a multiprocessor architecture, this measure defines the requirement on the inter-processor communication bandwidth. However, the latter is heavily dependent on the implementation, specifically, the hardware mapping of the algorithm.

In the following sections, the I/O bandwidth of the entire system, which is independent of the implementation, is used to gauge the communication bandwidth requirement of the algorithms. While the measure is simplistic, in many cases the measure correlates well with the actual bandwidth requirement of the algorithms.

2.1 Viterbi Detector

In recent years there has been a wide interest in high speed implementations of the Viterbi algorithm, which is an optimum algorithm for estimating the state sequence of a finite state process given a set of noisy observations [11, 36]. Driving applications include convolutional decoders for error correction, trellis code demodulation for communication channels, and digital sequence detection for magnetic storage devices in the presence of inter-symbol interference. In this section, an integrated Viterbi detector for (extended) Class 4 Partial Response Maximum Likelihood (EPRML and PRML) channels is described [95].

Figure 2-1 shows the block diagram of the Viterbi detector IC. It includes an equalizer

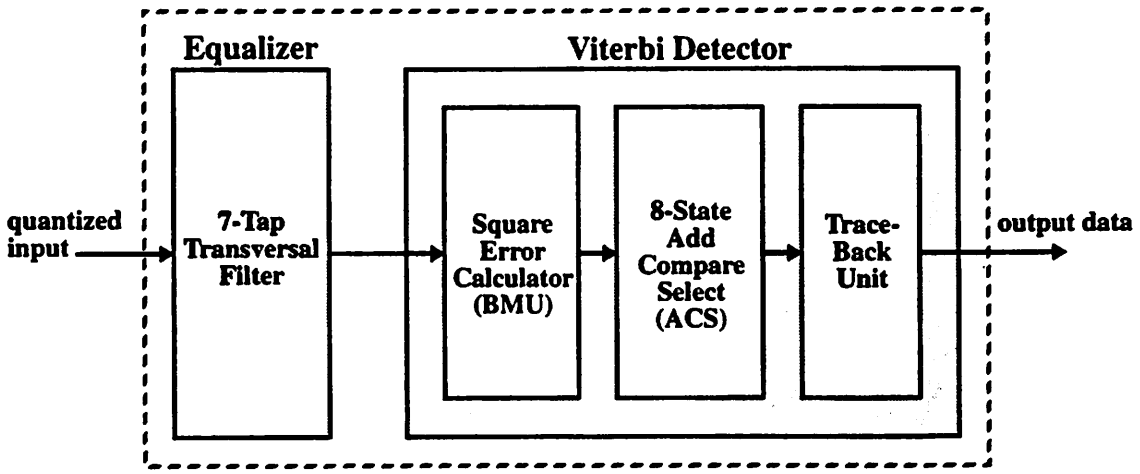


Figure 2-1: Block Diagram of PRML and EPRML Viterbi Detector.

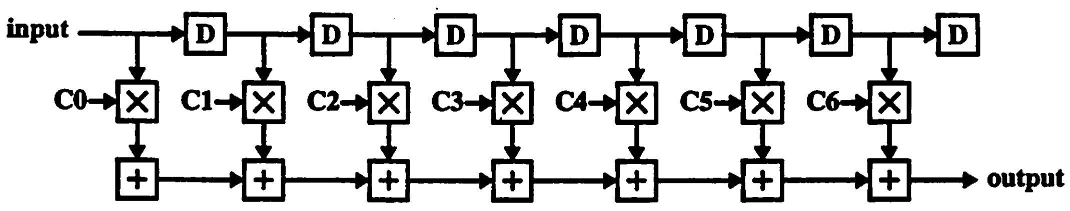


Figure 2-2: (a) Equalizer implemented as 7-Tap Transversal Filter.

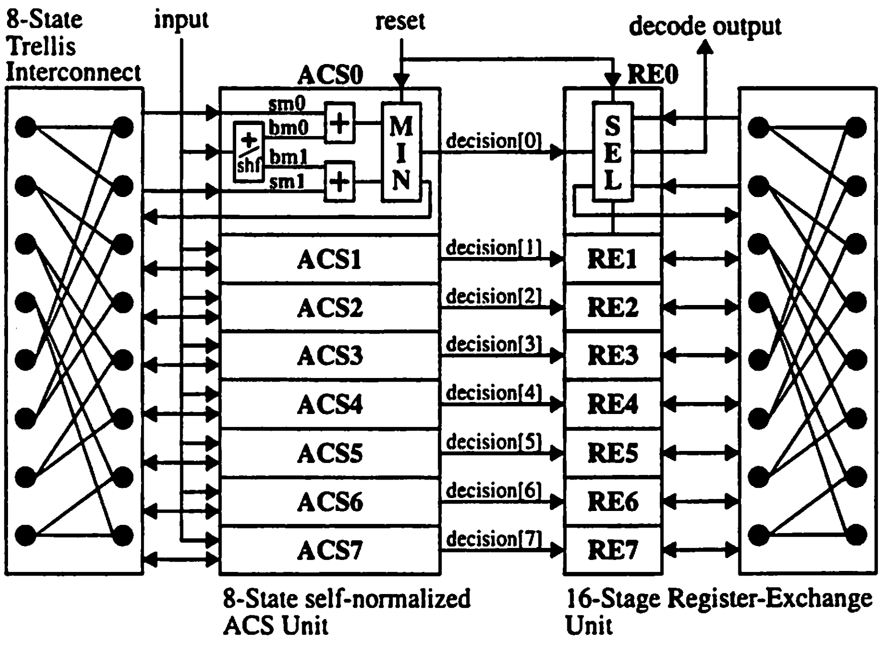


Figure 2-2: (b) Block Diagram of Viterbi Detector.

implemented as a 7-tap transversal filter, and a Viterbi detector with an 8-state Add-Compare-Select unit. Implementation examples of the equalizer block and the Viterbi detector block are shown in Figures 2-2 (a) and (b), respectively. The branch metric unit (BMU) calculates the branch metrics, which measure the likelihood of a state transition in the trellis diagram based on the square of the error between the observed input and the expected input. Associated with each state is a state metric which can be viewed as the accumulated metric along the *most likely* path leading to that state. The state metrics of the two predecessor states of each state are added to the two corresponding branch metrics and then the minimum of the two sums, corresponding to the more likely transition, is chosen to update the state metric for the next iteration. The decision is sent to the trace-back unit.

A Register-exchange network is used for the trace-back unit because of its simplicity. Associated with each state is a register which contains the survivor path leading to that state. In every cycle, one of the two registers of the predecessor states is chosen depending on the decision output from the ACS unit. The content of the register is then shifted by one bit with the decision output as the shift input and stored for the next iteration. The length of the register determines the *survivor path length*, typically about 5 times of the memory length of the encoder process. The bit shifted out from either one of the states, say state0, is the decoder output.

The reported ASIC, implemented in 0.8-μm CMOS, achieves a channel decode rate of 48 Mb/s running at 48 MHz. Because of the high speed, dedicated multipliers, adders are used for the transversal filter and a parallel ACS architecture is used.

About 4 GOPS of computation power is required. The dominant operations are multiplication, addition, shift, compare and select. While communications between the functional blocks are strictly feed-forward and systolic, very irregular communication patterns are needed inside the Viterbi detector for implementing the connection pattern of an 8-state trellis. I/O bandwidth requirement is acceptable as long as the ACS unit or the Register-exchange unit fit into a single-chip. Since a Register-exchange unit is used to implement the trace-back unit, no large memory is needed. However, if a long *survivor path length* is desired, it is more economical to use memories to buffer the decision outputs from the ACS unit for trace-back [11]. The control for the algorithm is very simple because of the dedicated nature of the functional blocks.

Despite the complexity of the Viterbi algorithm, high speed implementation is achieved because of the abundance of both spatial and temporal concurrencies. The equalizer is a feed-forward structure and can therefore be pipelined down to very low granularity. Performance is mostly limited by the single-delay recursion loops present in both ACS unit and the trace-back unit. Algorithmic techniques such as lookahead techniques [9] and block precessing techniques [10] can be applied to the Viterbi algorithm to expose more parallelisms and improve speed.

2.2 Hidden Surface Processor for 3-Dimensions Graphics

In 3-dimensional computer graphics, every polygon must be checked to determine whether it is visible or hidden from the viewer. A video processor, which eliminates hidden portions of surfaces using a Z-buffer algorithm on a raster scan-line basis, is described in this section [81].

A system block diagram of the hidden surface processing system describing the detailed operations of each pixel processor is shown in Figure 2-3. Each pixel processor handles a single pixel and 256 processors connected in a linear array are needed to process a 256-pixel scan-line. Tokens of line segments are fed into one end of the array and processed systolically. A segment token consists of:

- **X** starting x-coordinate of the line segment.
- **ΔX** length of the line segment.
- **Z** z-coordinate at the starting x-coordinate.
- **ΔZ** tilt of the line segment.
- **I** intensity of the line segment.

Xi is decremented as it propagates through the array until it becomes zero. When **Xi** becomes negative, $HSPP_i$ is at the beginning of the line segment. The subsequent HSPP's then decrement **ΔXi** until **ΔXi** also becomes negative. This indicates that the end of the line segment is reached. The enable flag, $\mathbf{enable}_i$, is asserted if it is within the line segment, i.e., when $\mathbf{X}_i$ is negative and

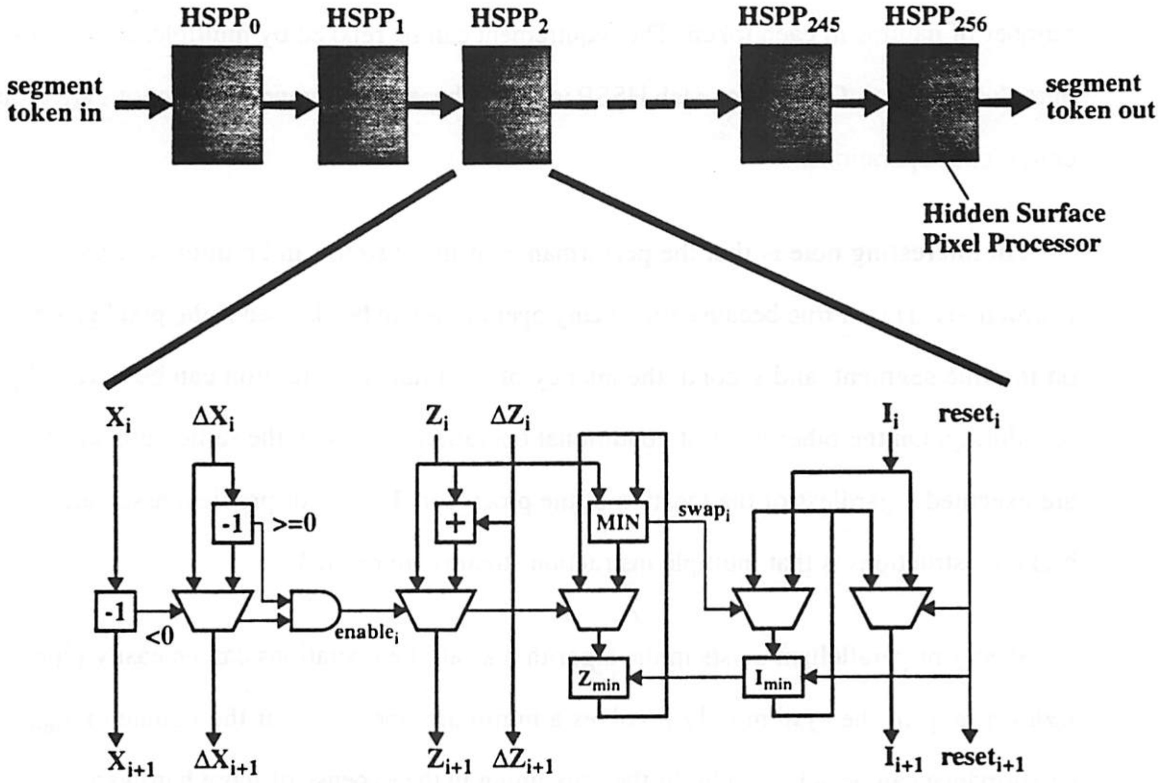


Figure 2-3: 256-Pixel Hidden Surface Pixel Processing System.

ΔX_i is not.

Upon receiving the enable flag, the processor updates the locally stored $\mathbf{Z_{min}}$ by comparing the incoming **Z** with $\mathbf{Z_{min}}$. It also computes the new z-coordinate of the tilted line segment at the next pixel by summing **Z** and Δ**Z**. If $\mathbf{Z_{min}}$ has been updated, a swap flag, $\mathbf{swap_i}$ is asserted and $\mathbf{I_{min}}$ is simply updated with the incoming **I**. After processing of all the segments in a scan line, a special *reset* token is sent down the array. This triggers the scan-out of the stored $\mathbf{I_{min}}$'s and resets $\mathbf{Z_{min}}$ and $\mathbf{I_{min}}$ so that processing of the next scan-line can begin.

The implementation reported can process one line segment token every 4 clocks at a 20 MHz rate for a 256-pixel scan line, corresponding to 5 millions line segments per second. On the order of 10 GOPS computation bandwidth is required. Only simple arithmetic and minimum operations are used. Data flow is predominantly systolic. High I/O bandwidth is needed because of the large number of datums in each token. The requirement can be relaxed by multiplexing several datums onto the same bus. Control for each HSSP is simple because all branch instructions are replaced by conditional operations.

An interesting note is that the performance of the system can be improved by using branch instructions. This is true because first, many operations can be skipped if the pixel processor is not on the line segment, and second, the latency of the branch instruction can be hidden by careful scheduling. On the other hand, if conditional operations are used, the same number of operations are executed regardless of the location of the processor. The major problem associated with using branch instructions is that multiple instruction streams are needed.

Plenty of parallelism exists in the algorithm as all the operations can be easily pipelined. The *tightest* loop of the system only involves a minimum operation for the update of Z_{min}. Higher performance can be achieved by further pipelining in the expense of more hardware.

2.3 High-Definition Digital Camera Processor

In this section, a fully digital video processor for HDTV cameras [60] is presented. The system block diagram of the video processor is shown in Figure 2-4. It has three 10-bit input and output channels (R, G, B) at 74.25 MHz and uses as much as 16-bit precision for internal calculations for higher accuracy. The processing functions include shading, aperture correction, color balance, gamma correction, and dynamic range compression. Block diagrams of the

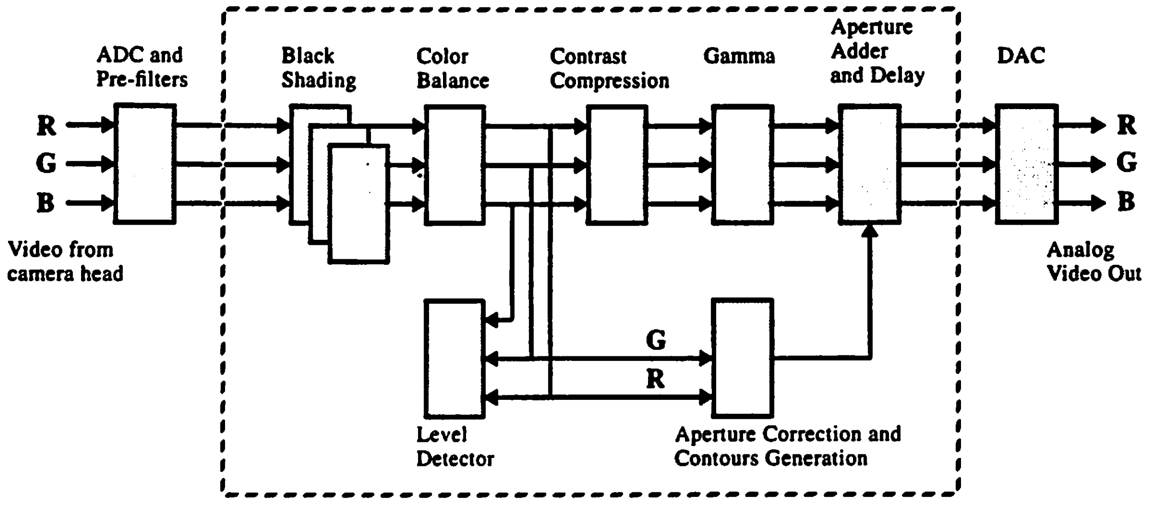


Figure 2-4: HDTV Digital Camera Processor.

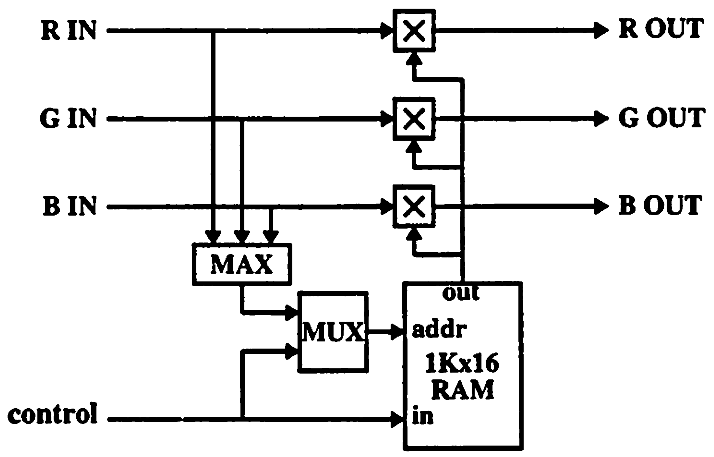


Figure 2-5: Dynamic Contrast Compression.

functional blocks are shown in Figure 2-5, 2-6 and 2-7. All the filter blocks and aperture processing blocks are transversal filters with a small number taps. The filter coefficients are fixed at run-time, but are important design parameters during algorithmic development. This application

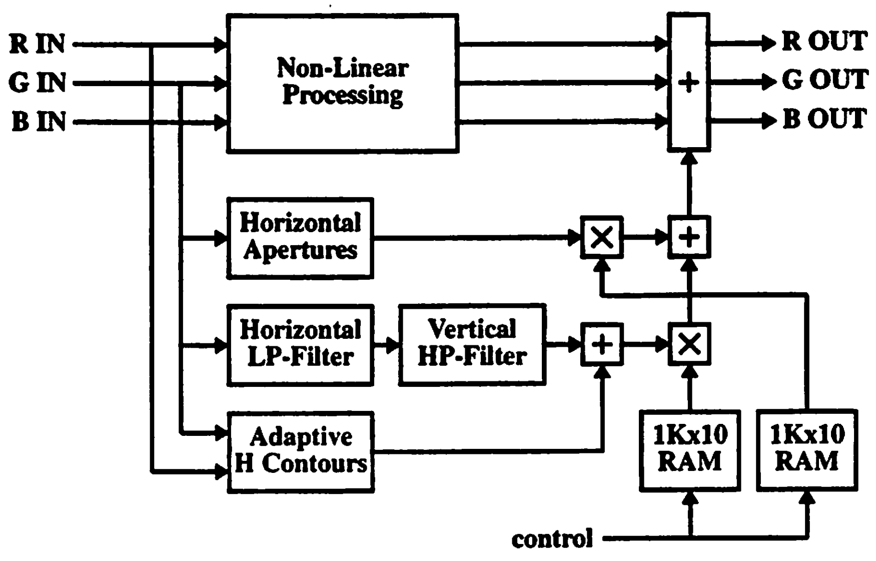


Figure 2-6: Edge Enhancement System.

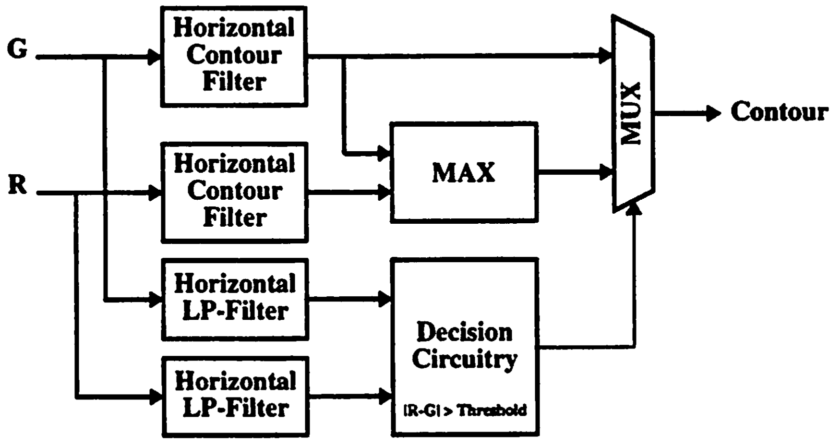


Figure 2-7: Adaptive Horizontal Contours.

is of particular interest because the types of processing performed are typical in many video processing applications.

At the pixel rate of 74.25 MHz, tenth's of GOPS are needed for each color component. Typical operations include multiplication, addition, maximum and minimum operations, and memory read and write. Only simple branches are needed, some of which can be replaced by conditional operations, e.g., select operations. Data communications are straightforward (almost systolic) between functional blocks, but are heterogeneous inside the functional unit. I/O bandwidth requirement is large because of the high data rate and multiple I/O channels. Shift operations may be used to multiplex the three 10-bit color components onto two 16-bit I/O buses to save I/O bandwidth.

Several small (a few kilobytes) random-access memories, used as lookup tables, e.g., for non-linear processing, are accessed in parallel at the high data rate, resulting in high memory bandwidth requirement. A few large sequential-access memories (up to hundred's of megabytes) are used as video frame delays and line delays. The implementation uses distributed parallel memories to handle the memory bandwidth bottleneck. Again, control is very simple because of the dedicated nature of the functional units. Control is needed mainly to distinguish the normal data processing phase and the idle phase (the vertical blanking period), during which new system parameters and lookup tables may be updated. As in the previous examples, plenty of fine grain parallelism is available in the system and pipelining can be easily applied to boost the performance.

2.4 Summary

In this section, common and pertinent characteristics of the algorithms discussed in the previous sections are summarized and analyzed below. Although only 3 representative algorithms are examined, the summary does give a general picture of the DSP algorithms in the application domain targeted by this research as follows:

- high computation requirement, on the order of 10 to 100 GOPS.
- moderate to high data rate, 10 to 100 MHz.

- high I/O bandwidth, 100 to 500 MByte/sec.
- varied requirements on memory bandwidth and memory capacity.
- both regular and irregular data flow.
- simple control functions.
- plenty of fine grain concurrencies, both temporal and spatial.

In general, the target DSP applications place tremendous demand on both computation and I/O bandwidth requirements. High data rate is needed, especially in video applications. Requirements on memory bandwidth and capacity vary depending on the individual applications. For instance, while the first two applications do not utilize any memory, the HDTV Camera system makes heavy use of memories, requiring up to a gigabyte per second memory bandwidth and hundreds of megabytes of memory capacity. While regular data flow pattern is common, a significant amount of heterogeneous data flow is present in general. Most applications are dominated by computation, requiring only simple looping and branching, and therefore simple control functions are sufficient. Abundance of both temporal and spatial concurrencies in the algorithms can be exploited by pipelining and parallelizing approaches.

CHAPTER 3

Hardware Prototyping Approaches

In this chapter, efficiencies and limitations of existing hardware prototyping approaches are first discussed, leading to the conclusion that an approach based on programmable, multiprocessing hardware can satisfy the performance and flexibility requirements most cost-efficiently. Programmable multiprocessor IC architectures for high throughput DSP targeted by this research are then classified according to the granularity of the basic processing elements and the control/arithmetic ratio. Representative architectures are cited, analyzed and compared. Special attention is given to the architectural domain where the proposed PADDI-2 architecture belongs.

3.1 Traditional Breadboarding

One of the mostly widely used hardware prototyping techniques is the traditional breadboarding approach in which hardware prototypes are built using off-the-shelf generic IC's. These pre-fabricated IC's range in complexity from TTL/ECL bit-slices and arithmetic units, to dedicated processors such as programmable transversal filter chips and video engines for specific applications [40, 86, 112], to high performance domain specific processors such as general purpose video signal processors [45, 101, 102]. The main advantage of this approach is that design and testing of custom chips are not required. Good performance can be expected if the parts needed for the specific application are available. Another important advantage is that manufacturers of these generic parts very often accompany the hardware with software development systems for the designers to ease system design.

Despite its popularity, this approach suffers from many shortcomings. Perhaps the greatest drawback of all is the lack of flexibility. The prototype, once built, cannot be easily changed to handle other design alternatives. Although some degree of programmability is usually available for the more complex application-specific processors, e. g., to configure the number of taps of a transversal filter chip, the programmability is usually too limited for the system designers, especially if the type of processing is altered drastically. Another equally important problem is that a complete system is usually built from chips of very different architectures and/or programming paradigms. The heterogeneity makes programming a very tedious task, e.g., custom designed hardware and software interfaces between chips are often needed. To implement systems that require high speed operations, custom designed printed circuit boards are often needed, further increasing the development cost and time.

As an example, the hardware prototype of the high-definition digital camera system described in section 2.3 was designed using this approach. The prototype was constructed using mostly ECL components to achieve real-time HDTV video processing. Despite the fact that the prototype was specially designed to handle the processing functions of HDTV video, it still could not fulfil the needs of the system designers who constantly desired more programmability and flexibility in order to experiment with new sophisticated algorithms [30].

3.2 Application-Specific Integrated Circuits

In order to achieve high performance, or in the absence of pre-fabricated IC's suited for the particular application in hand, some designers use application-specific integrated circuits (ASIC's) for hardware rapid prototyping [81, 82, 95]. The main advantages of this approach are higher performance and reduced chip count due to higher level of customization and integration. For example, several ASIC's were designed to build a connected speech recognition system capable of recognizing words out of a 60,000-word vocabulary [82]. Parts of the ASIC design may also be reused when IC's for the final product are developed.

Unlike the traditional breadboarding approach, the ASIC approach suffers from long turn-around time and high NRE costs mandated by the fabrication and design of custom IC's. Using gate-arrays for ASIC fabrication can shorten the turn-around time somewhat, depending on the process technology used, but some performance is sacrificed in general. ASIC's are more flexible than off-the-shelf components because programmability needed for the specific application can be *designed in* during chip design. However, the design still cannot be changed once the chip is fabricated and modifications are both costly and time-consuming. Sometimes the algorithm may change so drastically during system development that the ASIC can be rendered useless. In addition to the usual risk inherent in the design of a new custom chip, the overhead in chip testing and debugging always adds to the development time. Due to the custom IC design, special board-level design tailored to a particular application is often required.

Because ASIC design offers good performance and potential for system integration, there are widespread research activities aimed at improving the design time of ASIC's. Besides the conventional computer aided design tools for IC design, a very promising approach is to *synthesize* an ASIC from high-level design specifications [29, 31, 77, 83]. Not only is the design time reduced drastically, chip testing and trouble-shooting are also simplified because the circuits generated by the synthesis tool are *correct by construction*. For example, the synthesis tool called

HYPER [77] accepts specifications of DSP algorithms in high-level languages such as SILAGE [49] and VHDL [64], performs estimation, transformations, partitioning, assignment and scheduling, and generates silicon layout as the final product. Many researchers are also interested in extending the synthesis approach to the system level to achieve rapid prototyping of a complete system [96].

3.3 Programmable Multiprocessors

Perhaps the biggest drawbacks of the hardware prototyping approaches using generic IC's or ASIC's are the long development time and large NRE costs due to the needs of custom printed circuit boards and/or overhead associated with ASIC fabrication. To overcome these problems, many system designers have turned to programmable hardware as their prototyping platforms [8, 56, 78, 85, 107]. As an example, the computation intensive speech recognition systems described in [8, 85] achieve real-time capability by using parallel commercial processors such as the 8 MFLOP AT&T DSP32C floating point digital signal processor and the 10 MIPS Weitek XL-8032 processors. Using existing hardware, this approach minimizes NRE costs and avoids tedious testing and debugging of hardware. Moreover, development time is reduced to the time needed for programming the hardware. By migrating the tasks of prototype development from hardware to software, flexibility, high reusability, modularity and ease of debugging are ensured. It is clear that compared to the breadboarding approach and the ASIC approach, programmable hardware satisfies the stringent performance requirement and offers the needed flexibility most cost-efficiently.

Even with rapid advances in processing technologies, a single chip is not able to deliver the GOPS's of computation throughput typically required by our target DSP algorithms, and therefore, almost without exceptions, multiprocessing based systems are needed. The use of programmable multiprocessor systems for rapid prototyping of DSP systems presents many challenges, due to the facts that the high computation requirement of the target DSP algorithms demands a large number of processors, and that the high data rate puts tremendous pressure on inter-processor communication bandwidth. Memory bandwidth can also be the bottleneck in some applications.

How are large number of processing elements controlled and synchronized, SIMD or MIMD, centralized or distributed? What communication mechanisms and network topologies are appropriate for different classes of algorithms? What programming models should be used to ease programming and compilation? These issues, and many more, are the subjects of active researches [50]. Some of these issues will be revisited in Chapter 7, where the architecture of a system based on the PADDI-2 single-chip multiprocessor is described.

In this research, the focus is on the architectural design of the basic single-chip processing component, on top of which the complete multiprocessing system can be built cost effectively. Although it is obvious that chip-level architectural design is heavily influenced by the overall system architecture, there is still plenty of room for architectural innovations, since current IC process technologies enable integration of millions of transistors onto a small die, and further advancements can be expected.

We restrict our application scope to fixed-point DSP. Although floating-point DSP is much superior to fix-point in terms of dynamic range, fixed-point DSP is widely used because fixed-point hardware is both cheaper and faster than its floating-point counterpart. A good review of fixed-point DSP vs. floating-point DSP can be found in [62]. In the following sections, architectures of single-chip integer multiprocessors will be classified, analyzed and compared.

3.3.1 Architectural Classification

"A good classification scheme should reveal why a particular architecture is likely to provide a performance improvement."

David Skillicorn, *A Taxonomy for Computer Architectures*

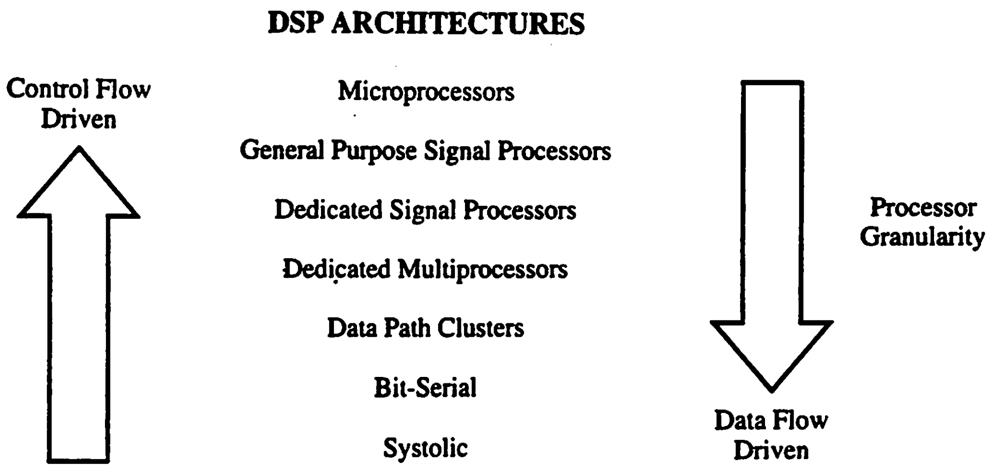


Figure 3-1: Architectural Classification based on Control/Arithmetic Ratio.

Besides understanding past accomplishments, classifying architectures helps reveal missing gaps in the architectural spectrum that might not otherwise have occurred to a system designer. Moreover, a good classification scheme should give strong hints why or why not a particular architecture is likely to provide performance improvement over its counterparts [91].

Many taxonomies for computer systems have been proposed: from the classical Flynn's taxonomy based on parallelism within the instruction stream and data stream [35], to the elaborate extension by Skillicorn [91], to the taxonomy restricted to IC architectures suggested by Keutzer [55]. A comprehensive review of taxonomy for DSP architectures can be found in Chen's thesis [27]. We found that the classification scheme for DSP architectures described in [14] by Brodersen and Rabaey is simple and yet very powerful and relevant to the classification of single-chip processor architectures.

In their classification scheme, DSP architectures are classified based on the amount of operation sharing on an arithmetic unit, as shown in Figure 3-1. One end of the spectrum represents the traditional microprocessor architecture, where all arithmetic operations are time-multiplexed on a single general purpose ALU. This architecture is classified as **control flow**

driven, since the functionality of the programmed device is completely determined by the contents of the control section. At the other end of the spectrum are architectures such as systolic arrays (bit-parallel or bit-serial), where operations are mapped to separate dedicated hardware units and seldom multiplexed. These architectures are called **hard-wired** or **data-flow driven** and the control section is minimal or nonexistent. In between, a complete suite of architectures such as dedicated processors and data path clusters can be found. *One of the major challenges in architectural design is to strike the right balance between control and data path sections for a given application and a given throughput range* [14].

This classification does not explicitly consider granularity of the processing elements. However, it is included implicitly because as the control/arithmetic ratio decreases across the spectrum, the complexity of the processing element can be expected to decrease also. Since we are mostly interested in single-chip implementations, we can carry the inference one step further to classify IC architectures based the number of on-chip processing elements (or execution units). This classification scheme, which is, in fact, very similar to that of Brodersen and Rabaey's, is illustrated qualitatively in Figure 3-2. Clearly, the absolute number of processing elements depends strongly on the implementation, e.g., the process technology used and the design style. However, the general trend is still valid as one goes from one architectural domain to another across the spectrum.

In Figure 3-2, the rough trade-offs between performance and flexibility are shown. As the number of processing elements increases, the performance can be expected to improve accordingly. It is true that techniques such as pipelining can boost performance without increasing the number of processing elements[1]. However, the same techniques can conceivably be applied to all the architectures across the spectrum, resulting in similar performance improvement and therefore the general performance trend still holds. Another way to comprehend the trend is to

1. A detailed discussion of pipelines in single-chip DSP processors can be found in [33].

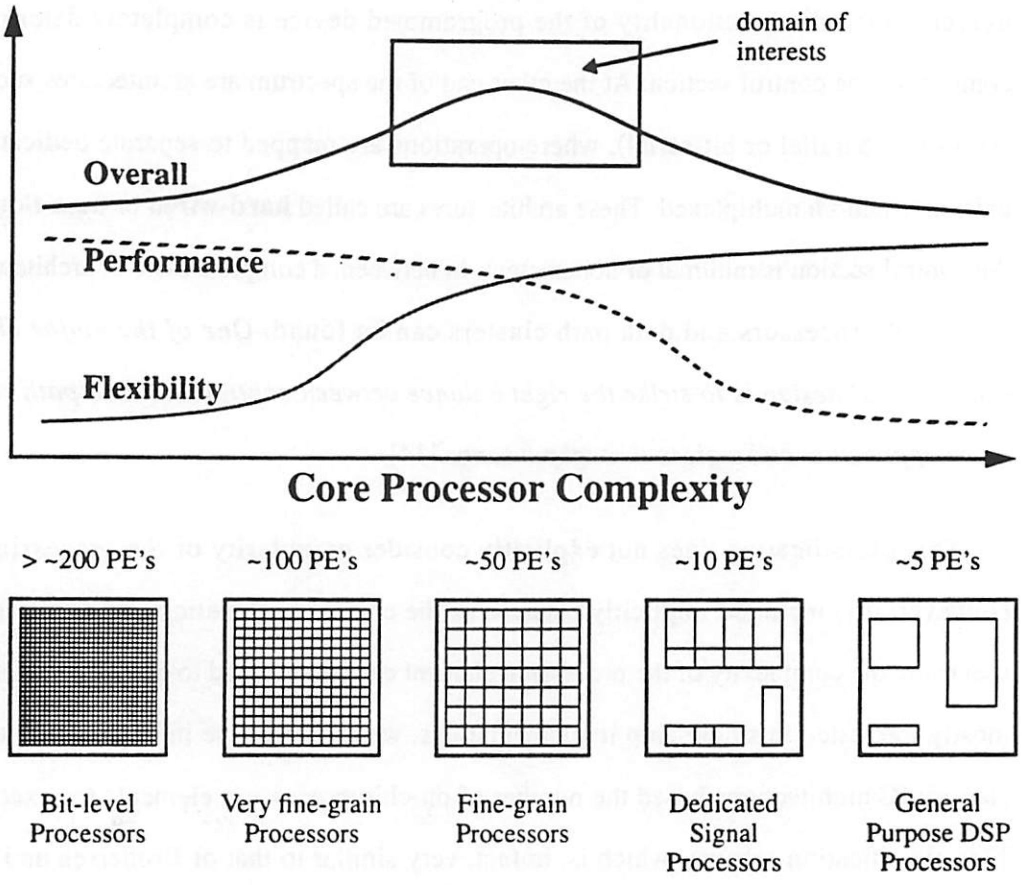


Figure 3-2: Architectural Classification based on Processor Granularity.

consider the complexity of the control section of the processing element. As the spectrum is transversed in the direction of decreasing processor complexity, the control section of the precessing elements become smaller and smaller. Eventually the control section can be eliminated entirely when the system is completely hard-wired. Simplification of the control section results in the availability of more silicon area for execution units and faster clock speed, and hence increased performance.

Unfortunately, the trend in flexibility is the reverse of the performance trend. While a control driven architecture such a microprocessor can be easily programmed for various tasks with the help of high-level language compiler, the programming of the data flow driven architectures such as a systolic processor array is much more involved. Not only do programs for a larger number of processing elements need to be written, inter-processor communications and synchronization have to be carefully handled before the processing elements can work in harmony to produce the correct results. Sometimes the communication network between the processors can limit the application scope of the architecture, e.g., systolic processor arrays, which only support communication between neighboring processors, can only be applied to a certain class of algorithms [58, 100]. The problems are further aggravated if heterogeneous processing elements are used. The degree of inflexibility worsens as the relative size of the control section to the data path section decreases, i.e., as the processor complexity is reduced.

The problem of rapid prototyping poses the interesting challenge of designing architectures that can achieve the high performance of the data flow driven architectures and yet exhibit the flexibility of the control driven architectures. As illustrated in the classification scheme in Figure 3-2, we conjecture that the overall optimum may lie in the middle of the architectural spectrum where fine-grain processor elements are used. In the following section, representative architectures in each region of the spectrum are analyzed to support our conjecture.

3.3.2 Programmable (Multi-)Processor Architectures

In this section, single-chip processor or multiprocessor architectures with various processor complexity are studied. As the number of processing elements increases, the impact of the communication network on the efficiency of the architecture becomes more and more significant, and therefore architectures are further differentiated by the flexibility of their communication networks. Representative architectures in each architectural domain, as tabulated in Table 3-1, are chosen for detailed discussions. Using the algorithms described in Chapter 2 as the target applications, these architectures are analyzed and evaluated, with special attention given to architectures using fine-grain processing elements. This is by no means an elaborate list, however, many of the arguments can be extended to similar architectures within the same domain. These

Processor Complexity	Coarse (~5 PE's)	Medium (~10 PE's)	Fine (~50 PE's)	Very-fine (~100 PE's)	Ultra-fine (> ~200 PE's)
Local Communications	-	-	DataWave [88, 89]	Data-driven Processor Array [57]	-
Flexible Communications	Commercial DSP Processors	NEC VSP [45]	VSP [92, 102]	PADDI [26, 27]	FPGA [39]

Table 3-1 : Single-chip Programmable (Multi-)Processor Architectures.

case studies provide important lessons and hints on how to design a good architecture for rapid prototyping.

Commercial DSP Processors

Since their introduction more than a decade ago, single-chip commercial DSP processors have remained the most dominant force in DSP. This is not at all surprising considering their good programmability, low cost and rapid and consistent improvement in performance over the years. Moreover, the conventional wisdom in the general-purpose processor business that VLSI advances are going to roll over everything is a very strong driving force. DSP processors, equipped with special architectural features to facilitate the tasks of signal processing, usually have higher performance than a general-purpose microprocessor in the DSP application domain. However, the distinction between DSP processors and general-purpose microprocessors has blurred in recent years, thanks to advances in both processing technology and architectures in the general-purpose microprocessor arena. For example, the recently announced UltraSPARC is able to support real-time video compression (H.261) and decompression (MPEG-2), an application domain formerly monopolized by DSP processors [93].

Perhaps the most celebrated feature of a DSP processor is the Multiply-Accumulate (MAC) instruction, which is used heavily in DSP algorithms. Some people gauge the performance of a

DSP processor simply by its MAC time. Due to the high memory bandwidth requirement, DSP processors usually adopt the classic Harvard architecture and use parallel on-chip and/or off-chip memory banks. Instruction execution is typically pipelined to boost clock speed and provisions are made to enhance the efficiency of loop execution, for example, the REPEAT instruction provided by some processors of the TMS320 family. Other important architectural features include rich addressing modes, e.g., bit-reverse addressing for fast-fourier transform and modulo-mode addressing for circular buffering, multiple dedicated address generation units, hardware I/O support such as timers, DMA controllers, etc..

The performance of a processor can be given by the average number of instructions executed per clock (IPC[2]) multiplied by the clock frequency. Given that modern processors usually adopt simple instructions and pipelined execution to increase clock frequency, the throughput is limited by the small IPC. There are two reasons for the small IPC: first, there may not be enough instruction level parallelism in the algorithm so that data dependencies and branch latency cause stalls in the execution pipelines, and second, IPC is limited by the moderate number of execution units in the processor. The latter is due to the fact that a general-purpose processor is primarily designed for time-multiplexing operations on a few execution units. Valuable silicon area, which could have been used for execution units to increase performance, is wasted on such extraneousness as complex control logic and large instruction cache. As an example, if a processor is dedicated to repeating the few instructions in a simple inner loop of an algorithm so as to satisfy the computation requirement, an instruction cache is not needed.

The Viterbi detector described in Chapter 2 is a good example to demonstrate the inefficiency of a DSP processor in exploiting fine-grain parallelism. About 80 instructions are executed to produce one decoded bit. Even if we make the very aggressive assumptions that there are enough

2. To evaluate performance of processors with different instruction sets, one must also consider the average amount of work performed by an instruction, i.e., complex instruction set vs. reduced instruction set.

instruction level parallelism in the code, i.e., there are no stalls due to data dependencies or branch latency, and that the processor can execute 4 instructions per cycle at the clock frequency of 100 MHz, it will still take 20 cycles to process one sample, resulting in a decode rate of a mere 5 Mb/s.

To achieve higher performance, designers must resort to multiprocessing. Some modern DSP processors have special architectural features to aid multiprocessing. For example, the TMS320-C40 from Texas Instruments Inc., are equipped with 6 high-speed serial links such that many common network topologies, e.g., hexagonal array, can be easily realized. However, the major bottleneck for developing systems using parallel DSP processors is often the lack of system-level software support. One exception is the transputer developed by Inmos, Inc. [51]. The entire transputer system architecture, from processor architecture, to instruction set, to programming language, is specially designed for multiprocessing. Using a concurrent language called Occam, which is based on the concept of communicating sequential processes, designers can easily program a set of processes to run on different transputers and to communicate with each other via *channels*, supported in both hardware and software.

NEC VSP: a Dedicated Signal Processor

Next in the spectrum is a class of dedicated DSP processors. Strictly speaking, these dedicated processors are not general-purpose processors because they are generally designed specially for a certain application domain, e.g., video processing. However, these processors are so flexible in terms of programmability that they can be applied to a wide range of algorithms within the application domain. Hence they are also known as application-specific processors. By focusing on an specific application domain, architectural features such as data path, memory organization and word width can be optimized more than a general-purpose processor. Consequently, a dedicated processor typically contains a higher number of processing elements or execution units, thus providing higher performance than a comparable general-purpose DSP processor.

A good example of a dedicated DSP processor is the super-high-speed video signal processor

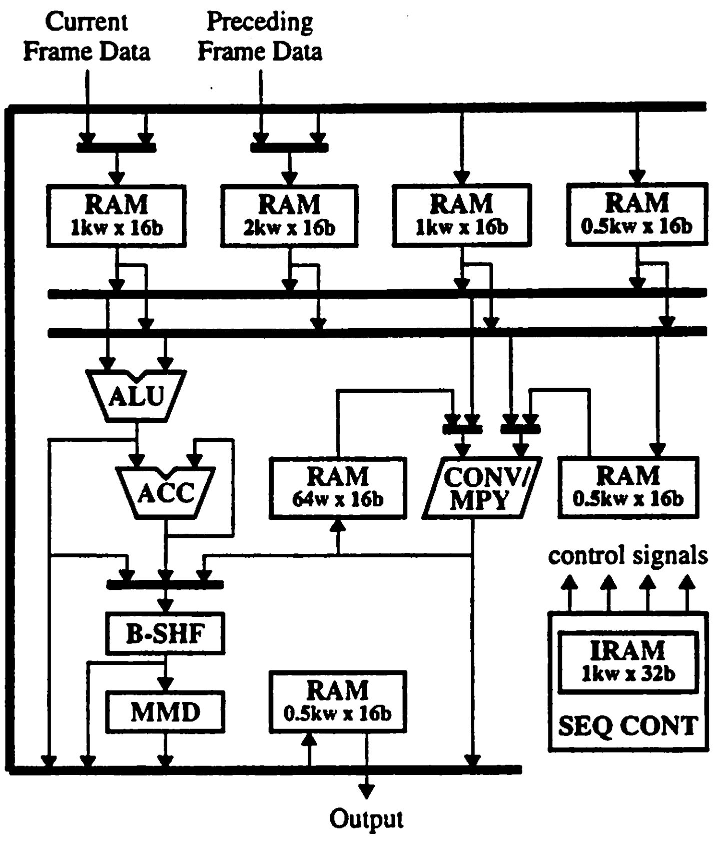


Figure 3-3: Simplified Block Diagram of NEC Video Signal Processor.

(VSP) from NEC Corporation [45], which contains many architectural features typical of dedicated processors such as a large number of execution units, optimized data paths, parallel memories, and microprogram based control. As shown in the block diagram in Figure 3-3, the NEC VSP contains two main execution units: a convolver/multiplier and a data path consisting of dual ALU's, a barrel shifter and a minimum/maximum value detector optimized for video signal processing such as inter-frame difference and motion compensation. Seven on-chip data RAM's, most of which are dual-ported, are used to sustain high data bandwidth to and from the speedy execution units. A total of twelve address generators, one for each port of the RAM's, run independently in co-processor mode to reduce instruction bandwidth from the central microprogram based sequencer.

Using a 0.8-μm triple-metal BiCMOS process, the chip runs at a 250 MHz internal clock frequency, and achieves 500 MOPS for convolution and 250 MOPS for multiplication and other arithmetic operations. For example, an 8x8 2-dimensional DCT can be performed using a single chip in 16.8 μs, equivalent to a pixel rate of 3.8 MHz.

Although the performance of dedicated signal processors is generally superior to that of general-purpose DSP processors, it is still not enough to satisfy the throughput requirement of our target algorithms. Due to the typical use of concurrently operating hardware units, programming is not easy because thorough understanding of the architecture is needed in order to orchestrate the units to work together correctly and efficiently. Moreover, because of the restrictive application scope, the manufacturers are likely to allocate less resources to compiler development, making programming even more difficult. The programming issue is further aggravated if multiprocessing is applied to improve performance. Another problem with using dedicated signal processors for rapid prototyping is that since it is optimized for a specific application domain, the processor may not be efficient for general DSP algorithms. For example, it is not clear how well the NEC VSP can handle the many branch instructions needed in the Viterbi detector and hidden surface processor described in Chapter 2.

DataWave: a Data-driven Array Processor

The DataWave [88, 89] belongs to a class of multiprocessor architectures called Wavefront Array Processors proposed by S.Y. Kung [58]. The wavefront array is an extension of the systolic array originally proposed by H. T. Kung [37]. Both systolic array processor and wavefront array processor achieve high performance through parallelism and pipelining using regularly, locally interconnected processing elements. However, the application is limited to a certain class of algorithms, such as regularly iterative algorithms [84] or the more general piecewise regular algorithms [100]. Examples are common in the areas of signal and image processing such as vector and matrix operations, sorting, and pattern recognition.

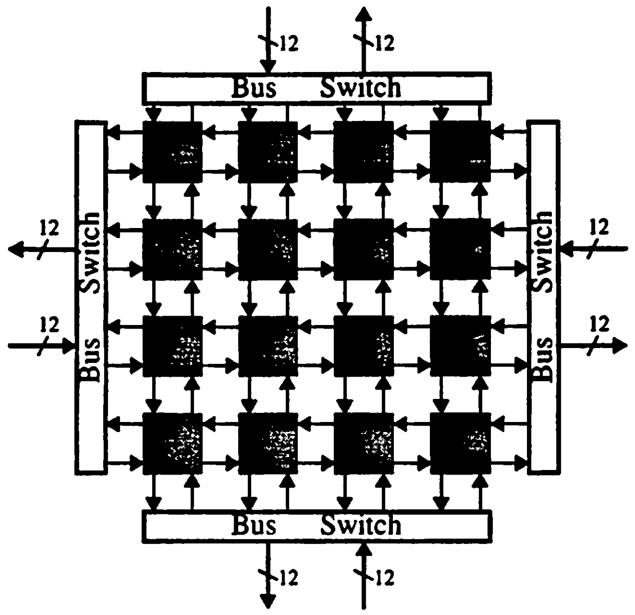


Figure 3-4: Block Diagram of a 16-Processor DataWave Chip.

The key difference between the two is that while all the operations in a systolic array are synchronized by a global clock, the wavefront array is asynchronous in that only the correct sequencing, not timing, is required. Hence, higher performance can potentially be achieved by the wavefront array by taking advantage of the differences in speed of the operations in the array. A serious problem with these array architectures, which is often overlooked by the designer, is that specially designed I/O units are often needed to provide enough data bandwidth to sustain the high performance [105].

As shown in the block diagram in Figure 3-4, a DataWave chip consists of 16 mesh-connected processing elements. Each processor independently executes a program stored locally, propagating data through the array in a wavefront-like manner. Instruction execution is data-driven, i.e., it is halted if any operands required are not available or if any inter-processor communications are blocked. Processing elements on the edge of the chip can communicate with another DataWave chip via the bus switch by time-multiplexing.

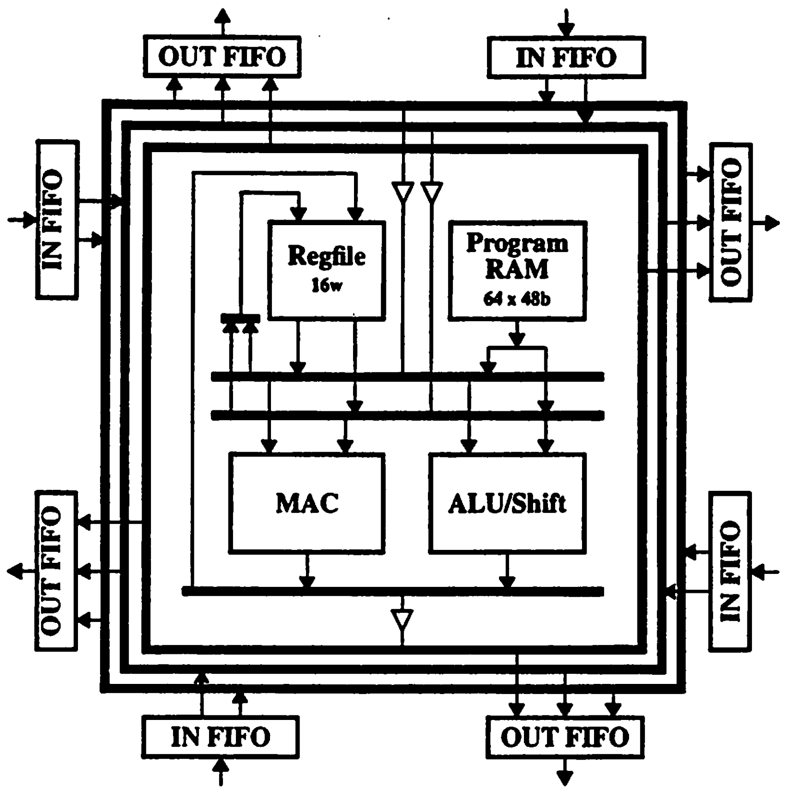


Figure 3-5: Block Diagram of a DataWave Processing Element.

Each processing element consists of a 4-port 16-word register file, an ALU, a multiply-accumulate unit, a 64x48b program store, and eight 8-deep FIFO's interconnected by three ring buses (Figure 3-4). Data path width is 16 bits with the exception of the accumulator which is 29-bit wide. A 5-stage pipeline is used to achieve a fast internal clock rate of 125 MHz in a 0.8-μm 2-metal CMOS process technology. An ALU operation and a MAC operation can be launched every cycle, resulting in a total peak performance of 4 GOPS. Due to the deep pipeline, branch latency is 3 cycles. The input FIFO and output FIFO on each side of the cell smooth out data transmission between cells and implement an asynchronous interface to eliminate a global clock and the associated clock skew problems. A complete software environment including a compiler for static data flow programs and a graphical simulator was developed for the architecture to ease programming.

This architecture is of particular interest to us because the architecture, though developed independently, shared many commonalities with our proposed architecture, e.g., the use of homogeneous fine-grain processing elements for high throughput, data-driven execution and static data flow programming paradigm. Another interesting feature of this architecture is the use of asynchronous interprocessor communications to achieve scalability both physically and architecturally.

Although the DataWave has the appearance of a Wavefront Array Processor and can certainly function like one, it is intended for a wider range of video applications, not limited to the traditional application scope of a Wavefront Array Processor. For instance, non-local communications can be realized by routing data through the PE's, which can perform arithmetic operations and at the same time move data from an input FIFO to an output FIFO. Nonetheless, when tackling algorithms with irregular data flow, many PE's, which are fairly complex, are likely to be under utilized. Moreover, the long latency due to data propagation through processing elements may not be acceptable especially for applications with tight recursive loops[3].

For example, while the mesh array topology can perform reasonably well for the hidden surface processor in section 2.2, it is not clear how well it can handle the many branch instructions and heterogeneous communications required in the Viterbi detector described in Section 2.1. Furthermore long latency is bound to hurt performance because of the single-delay recursive loop in the Add-Compare-Select unit. Finally, the DataWave architecture implements a computation array without any memory elements except for the small local register file. It will be interesting to see how memories are integrated into the system for memory intensive DSP applications.

3. Sometimes algorithmic transformation techniques such as the block processing technique can be applied to relief the recursive bottleneck [75].

Philips Video Signal Processor

The Video Signal Processor (VSP), developed by Philips Research Laboratories, is a general-purpose, programmable processor specially designed for efficient processing of real-time video signals using parallel, fine-grain processing elements [92, 102]. As shown in the block diagram in Figure 3-6, a VSP contains a total of 28 parallel processing elements: twelve Arithmetic Logic Elements (ALE's), four Memory Elements (ME's), six Buffer Elements (BE's) and six Output Elements (OE's), all of which are interconnected by a 28 by 60 by 12-bit crossbar switch matrix. Each processing element is connected to the switch matrix by one or more programmable-delay registers called *silos*. Implemented as 32 by 12-bit dual-port memories, the silos can realize sample delays, equalize pipeline delays, and are essential for achieving efficient cyclo-static program schedules.

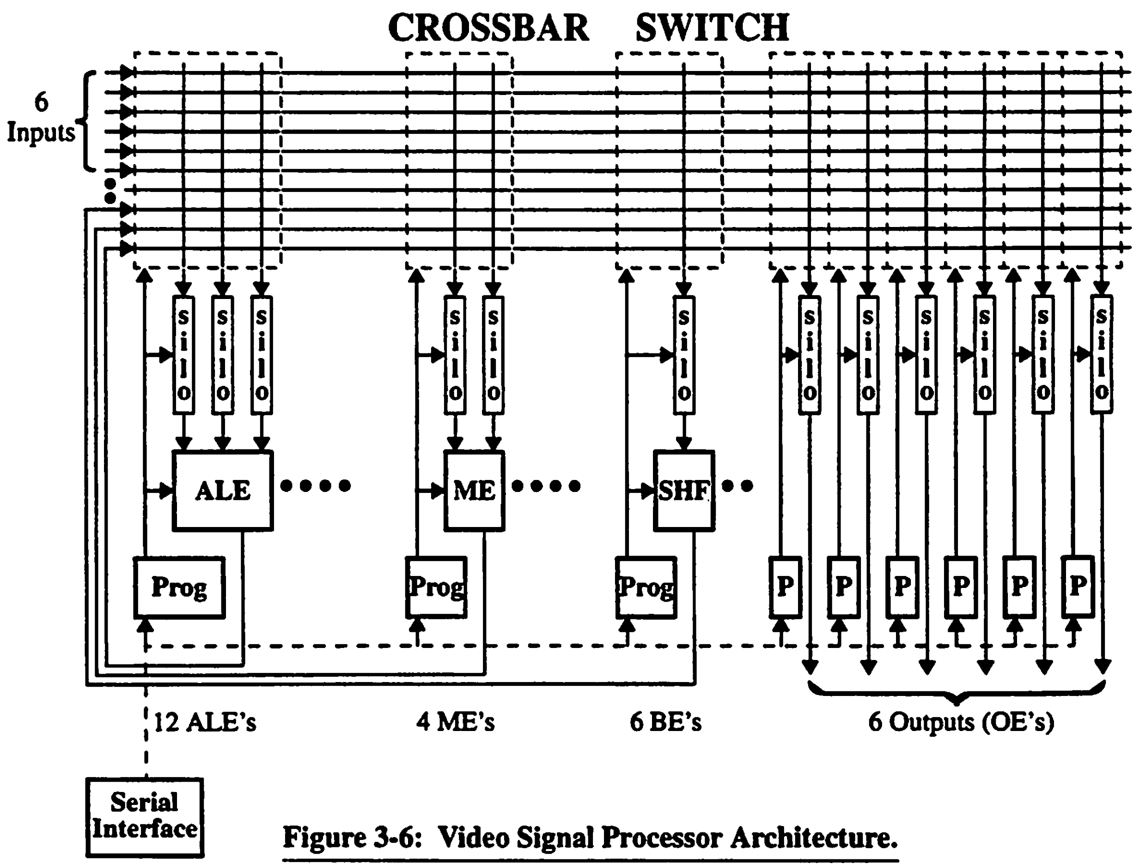


Figure 3-6: Video Signal Processor Architecture.

Using the cyclo-static scheduling algorithm, the execution of instructions in the parallel processing elements can be scheduled to satisfy certain criteria such as optimal processor utilization. Overhead of multiprocessing such as synchronization is minimized because each processing element simply cyclically executes the instructions stored in its local 32-word program memory according to the schedule. Hence the VSP is conceptually a Very-Long-Instruction-Word (VLIW) processor. Conditional execution is provided to replace data dependent branch instructions, which are not allowed.

An ALE consists of 3 silos, 3 barrel shifters and a 12-bit ALU, and is capable of performing arithmetic, logic, compare and shift operations. Multiplication is implemented sequentially by the partial multiplication operations based on Booth encoding. The ME contains a 2k by 12-bit two-port SRAM, 3 silos and some logic for address calculation, and can perform a read and a write operation concurrently every clock cycle. A BE, consisting of a silo and a barrel shifter, is mainly used a *floating silo* in series with any other processing element when longer length of the silo is needed. In addition, the BE can perform shift operations thereby freeing an ALE. Running at a clock frequency of 54 MHz, the chip provide 1.5 GIPS peak, and the six OE and six input buses provide a combined I/O bandwidth of 7.7 Gb/sec.

The VSP architecture is very interesting in that instead of developing a compiler for an architecture, the architecture is specially designed for the cyclo-static scheduling algorithm and many features such as the silos are included solely to enhance the efficiency of the compiler. Although cyclo-static scheduling allows multiprocessing with little overhead, the exclusion of data dependent operations may be too restrictive in general. For example, as discussed in Section 2.2, the performance of the hidden surface processor can be increased if branch operations are used instead of conditional execution. Moreover in order to obtain a *feasible* cyclo-static schedule, a large number of delays may be needed, resulting in the large number of silos and BE's totaling 23 Kb of two-port memory.

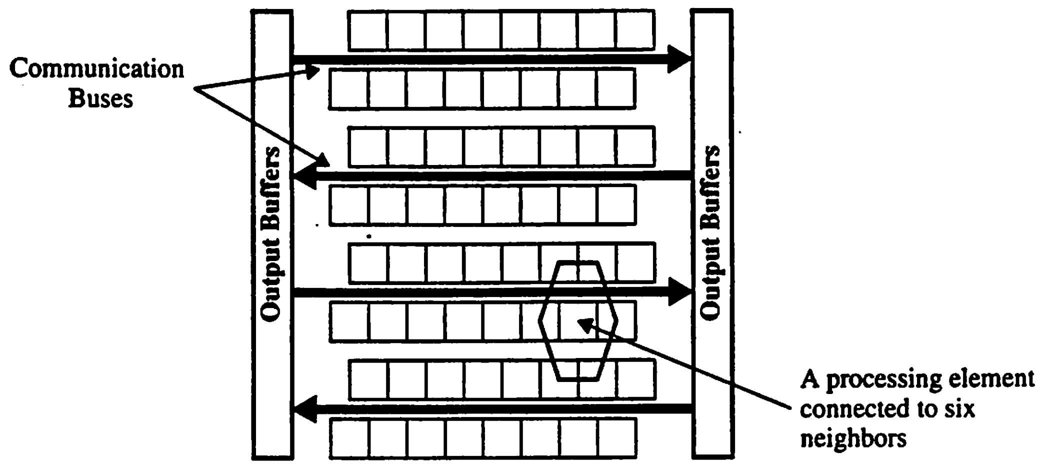


Figure 3-7: Block Diagram of the Data-Driven VLSI Array Processors.

Data-driven Processor Array

In this section, we will examine a data-driven VLSI array developed at the University of Massachusetts at Amherst [57]. The architecture also belongs to the class of wavefront array processors and bears many resemblances to the DataWave architecture discussed before. The key differences are that the data-driven VLSI array is specially designed for arbitrary algorithms and that it utilizes processing elements of much finer grain.

Figure 3-7 shows the block diagram of the processor array chip, which consists of an array of hexagonally connected arrays supplemented by several global communication buses. The basic idea is to map an arbitrary algorithm represented by a data flow graph to the processor array as shown in Figure 3-8. Using very simple processing elements, spatial as well as temporal concurrencies in the algorithms can be exploited through parallelism and pipelining to achieve high throughput.

The simplified block diagram of a basic processing element (PE) is shown in Figure 3-9. It contains six communication registers, an execution unit, a communication control block, a flag

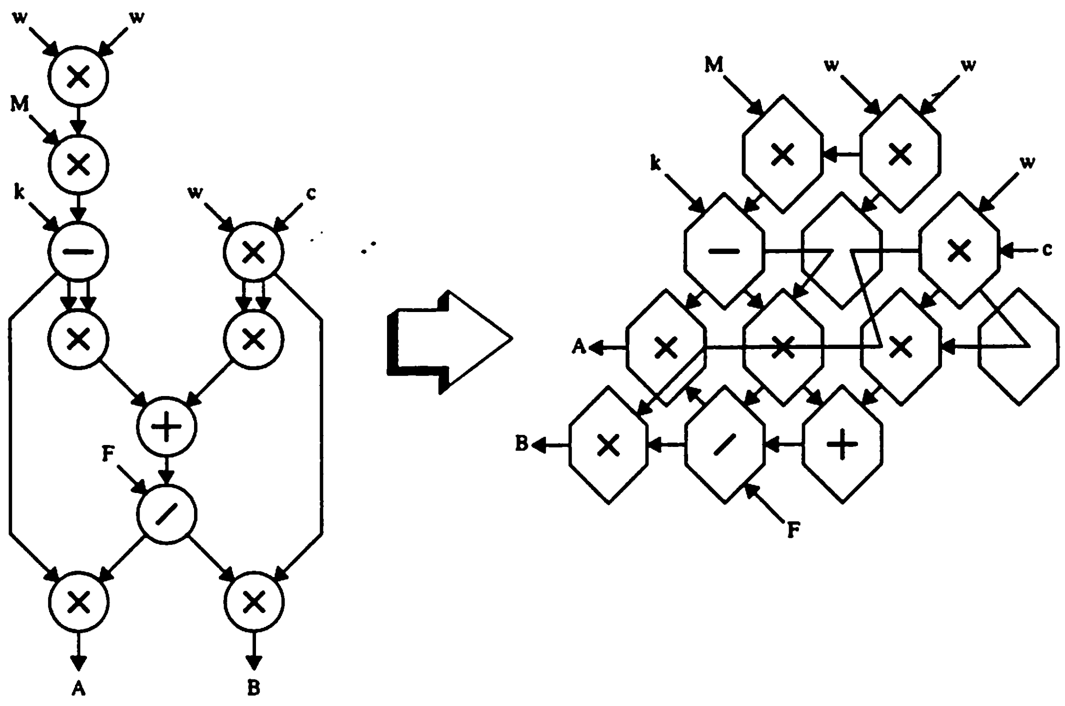


Figure 3-8: Mapping of a Data Flow Graph onto a Hexagonally Connected Array.

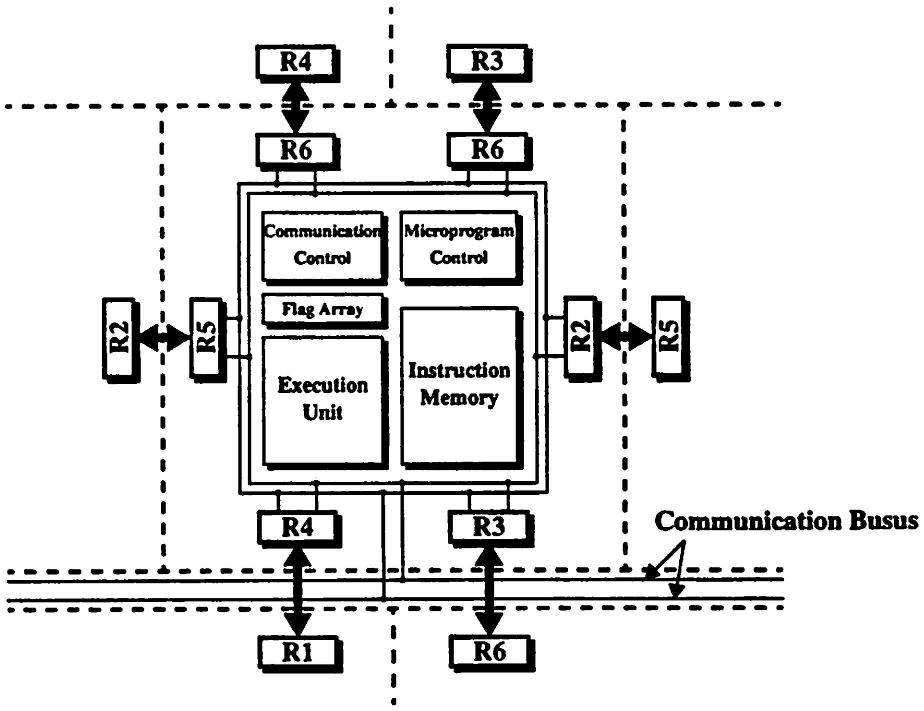


Figure 3-9: Simplified Block Diagram of the Processing Element of the VLSI Array.

array, a very small instruction memory and a microprogram control block. The communication registers, positioned in the periphery of the PE, buffer communications to the six neighboring PE's. The execution unit comprises a shift-register and an arithmetic logic unit capable of executing all common boolean and arithmetic operations. Shift, multiplication and division operations are performed sequentially. A carry bit is generated to allow multi-byte operations. The microprogram control block decodes instructions stored in the 6-word instruction memory into sequences of control signals. The flag array monitors the presence of operands to control data-driven operation of the PE. The width of all data paths and buses are 8 bits.

It is not clear if an operating chip has been designed, but the authors in [57] estimated that a chip containing about 100 PE's is practical in today's VLSI technology. Noteworthy architectural features include the global communication buses to facilitate I/O, especially to the middle of the array, and the use of PE's for routing to handle irregular data flow pattern. To reduce the overhead of data routing, the data-driven array, like the DataWave, uses long instruction words to allow execution of operations in parallel with data movement between ports. Because of the simpler PE's used, the inefficiency due to data routing by PE's is relatively less compared to the DataWave architecture.

Despite the use of global buses to enhance I/O, the local communication between PE's seriously degrades the efficiency of the architecture, as evident in the low average PE utilization of 27% or 38% reported, depending on the algorithms used to map algorithms to the array [71]. Similar to the DataWave architecture, the architecture can suffer when tackling algorithms consisting of recursions, due to the long communication latency. Another important problem is the low throughput of the frequently used multiplication operation due to the sequential execution. Furthermore, there is no provision to increase throughput, e.g., by executing multiplication in a pipelined fashion among several PE's.

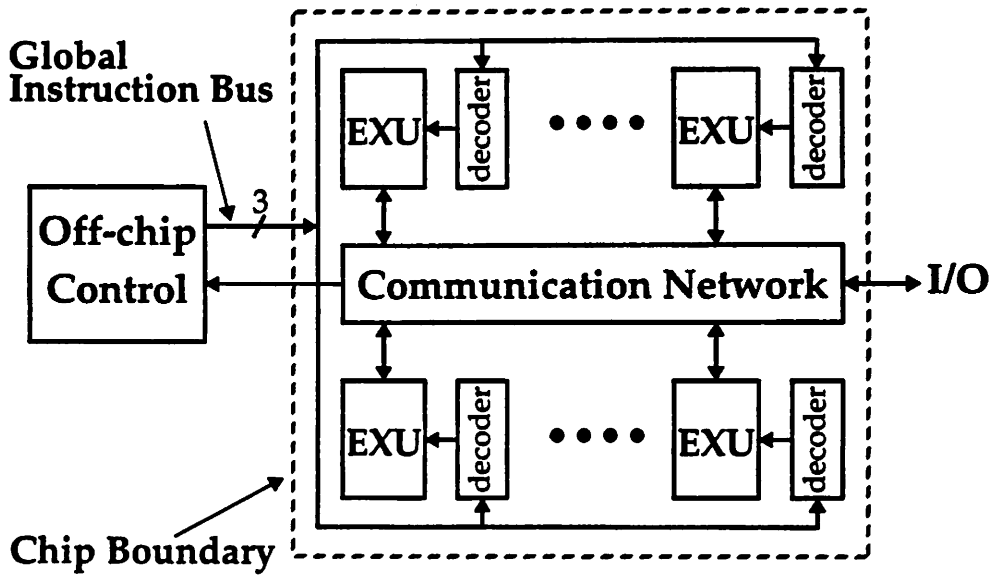


Figure 3-10: Block Diagram of PADDI Architecture.

PADDI

PADDI, an acronym for Programmable Arithmetic Device for DIgital signal processing, is an architecture previously developed by Chen of U.C. Berkeley for rapid prototyping of DSP [25, 27]. Its main target is high speed data paths for DSP using fine-grain processing elements. An understanding of the efficiencies and limitations of the architecture influenced many early ideas of this research.

As shown in the block diagram in Figure 3-10, a PADDI chip contains many fine-grain execution units interconnected by a flexible, dynamically configured communication network. Each execution unit has a local decoder that decodes a global 3-bit instruction broadcasted by an off-chip controller. The architecture is therefore basically a single-instruction stream multiple-data stream (SIMD) architecture with a very long instruction word (VLIW).

The PADDI instruction set includes shift and arithmetic operations, but logic instructions are not implemented. While variable-and-variable multiplication is not directly supported, variable-

and-constant multiplication is realized efficiently by shift-and-add operation. Data path width is 16 bit and two execution units can be linked to provide 32 bit operation at the expense of slower clock rate. Another interesting feature is that the 8-word register file of the execution unit can also be configured as a variable-length delay line for easy handling of sample delays in DSP algorithms. Running at 25 MHz, the prototype PADDI chip consisting of eight execution units can provide 200 MOPS using a 1.2-μm 2-metal CMOS process technology. More details about the PADDI architecture can be found in Chen's thesis [25].

From our experiences with the PADDI architecture, we found that the centralized control strategy employed in PADDI can sometimes be rather restrictive. For example, it is difficult to deal with several concurrently operating units with a single control thread, especially when data dependent operations are involved. The global control also makes it hard to scale the architecture to tackle more complex problem. On the other hand, the use of very fine-grain processing elements and flexible communication shows very good potential in achieving high performance for a wide range of DSP algorithms.

Field-Programmable Gate Array

In recent years, there has been a tremendous growth in a new breed of devices known as Field-Programmable Gate Arrays (FPGA's) [39]. Formerly in the supporting role of *saving glue logic* in the earlier days of their existence, FPGA's are now both powerful and cost efficient enough for system applications, thanks to rapid advances in IC processing technology and innovations in FPGA architectures. Usable gate counts up to 20,000 and system clock rates up to 100 MHz are possible with the current generation of FPGA's and higher density and speed can be expected in the future generations.

The device provides a new computing paradigm in that hardware can be reconfigured to execute an algorithm in its most natural form instead of the indirect process of first translating the algorithm to a format (machine code) understood by the hardware, and then interpreting the format

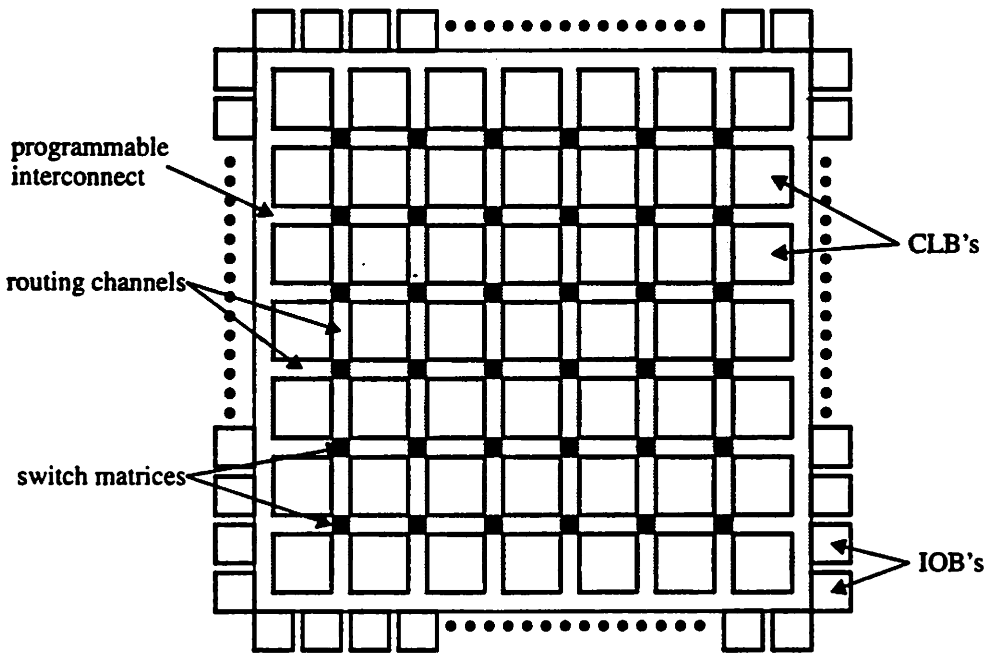


Figure 3-11: Basic Structure of XC4000 FPGA.

improve silicon utilization and usable gate count. Anti-fuse technology and channel-less routing can be applied to increase routability and density, and to shorten propagation delay of interconnect. Other innovations include the use of lookup tables in CLB's as small random-access memories and support for wide and fast decoders.

By providing reconfigurable hardware, flexible interconnect, and field-programmable ability, FPGA's offer excellent flexibility and quick turn-around time and are widely used for rapid prototyping of DSP and computer systems [18, 46, 47]. Note that many of these architectural features are also supported by our proposed architecture. However, because of FPGA's bit oriented architecture, its granularity may be too low for efficient implementation of wide data paths. When used to implement algorithms dominated by word-parallel operations, as is the usual case for DSP, an FPGA is likely to suffer from area inefficiency and speed degradation, compared to architectures tailored for word-parallel operations, given the same process technology and design style. Moreover, with limited routing resources, FPGA's can run into congestion problems when by hardware such as a processor to execute the algorithm. In short, the device provides the designers a *programmable* approach to design custom hardwired VLSI circuits. The hardware configurability can provide not only flexibility but also potential performance improvement, e.g., in a project at the Super-computing Research Center, an FPGA-based coprocessor called Splash beat a Cray-2 super computer by a factor of 330 on a DNA sequencing application [16].

The popularity of the device has fueled many research efforts in the areas of reconfigurable architectures, synthesis, logic optimization and mapping for FPGA's and applications [47]. New workshops have been established to facilitate exchanges of ideas and publication of results[4]. A good overview of the technology, architecture and CAD tools for programmable logic devices can be found in [17].

A typical FPGA architecture is examplified by the XC4000 architecture, one of the most popular FPGA families from Xilinx, Inc. [110]. As depicted in Figure 3-11, the basic structure of the XC4000 FPGA contains an array of configurable logic blocks (CLB's) and peripheral input/ output blocks (IOB's) interconnected by programmable interconnect in the form of routing channels and switch matrices. Figure 3-12 shows a simplified block diagram of an XC4000 CLB, which consists mainly of three lookup tables (or function generators), two flip-flops and several multiplexors. With thirteen inputs and four outputs, the CLB can be configured to implement fairly complex logic, e.g., a 2-bit full adder can be built using just one CLB. The IOB serves as interface between the internal logic and the external world and can be configured, e.g., as an input, output or tri-state pad, among the many options. The functionality of the FPGA is defined by the configurations in the CLB's and IOB's, and the interconnect setting, all of which are initialized at system start-up and stored in SRAM cells.

The architecture of FPGA is evolving rapidly. New devices may have simpler CLB's to

4. Examples are the ACM International Workshop on Field-Programmable Gate Arrays and the IEEE Workshop on Reconfigurable Architectures.

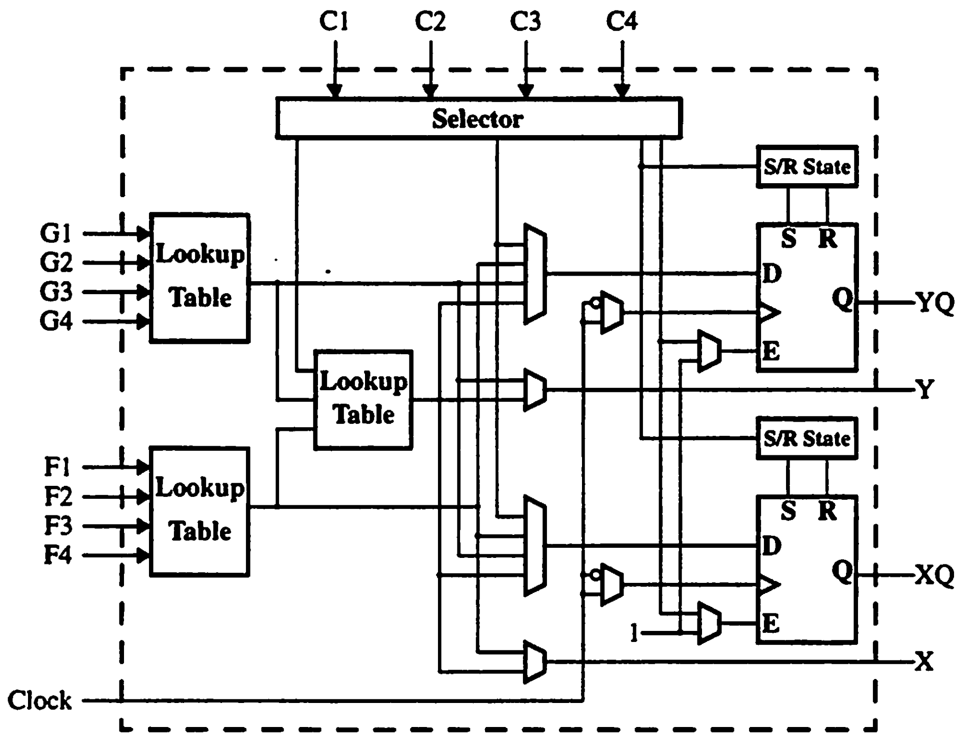


Figure 3-12: Simplified Block Diagram of XC4000 Configurable Logic

routing wide buses, hence jeopardizing CLB utilization and increasing propagation delay of the interconnect. More limitations of FPGA's for bus oriented applications are discussed in [25].

3.4 Summary

The traditional breadboard prototyping approach of putting together a system using off-the-shelf parts relies on the availability of chips with the desired functionality and performance. The resulting systems are often limited in programmability and inflexible to modification. Worst of all, the use of chips with heterogeneous architectures and programming paradigms makes programming and interface design a very tedious task. The ASIC approach allows high integration and improved performance, but suffers from long turn-around time, high NRE costs, and the risk of chip failures.

Hardware prototyping using a programmable multiprocessor provides the most attractive solution of all by eliminating error-prone hardware designs and offering low development cost, high flexibility, reusability, modularity, and ease of debugging (in software). Representative architectures of single-chip multiprocessors, classified according to the granularity of the basic processing elements, were presented and analyzed earlier in this chapter.

Von Neumann style general purpose microprocessors and DSP processors are flexible and easy to program because of the availability of high-level language compiler. However, these processors are limited by the heavy investment in control functions to enable hardware sharing, and therefore they are not efficient for exploiting fine-grain parallelism and cannot satisfy the high throughput requirement of our target algorithms. Multiprocessing using these processors can improve throughput, but the lack of software and/or hardware support can be the Archilles's Heel in the development of such systems. Moreover, the resulting system is still not cost-effective because a large number of chips are still needed to meet the throughput requirement.

Dedicated signal processors or domain-specific processors often achieve superior performance compared to the general purpose DSP processors by optimizing the architecture for a specific application domain. Unfortunately, the highly parallel architecture commonly used sacrifices ease of programming and compiler support is often limited, especially for multiprocessor development. Furthermore, the optimized architecture can jeopardize the efficiency of the processors for general DSP applications.

The DataWave architecture achieves very high performance using heavily pipelined fine-grain processor elements, but the inflexibility of the local communication networks degrades its efficiency for applications that require irregular data flow. A very similar approach, except for the finer granularity of the basic processor element, is taken by the Data-driven VSLI array and therefore it also suffers from the same inefficiencies as the DataWave architecture. Based on the cyclo-static scheduling algorithm, the Philips VSP enables multiprocessing with very little overhead, but cannot handle data dependent operations in general. The PADDI architecture has many interesting architectural features that enhances the performance for a wide range of DSP algorithms. However, the centralized single-instruction stream multiple-data stream (SIMD) control strategy limits the scalability and flexibility of the architecture.

Field-Programmable Gate Arrays (FPGA's) achieve quick around-time and excellent flexibility by offering field-programmability down to the gate level, but their bit-oriented architectures are area-inefficient and slow for implementing wide data paths commonly used for DSP applications.

Based on the arguments in this chapter, it is clear that no existing architecture can provide both the flexibility and the performance needed for rapid prototyping of our target DSP algorithms in a cost-efficient manner. In the next chapter, we propose an improved architecture called PADDI-2 specially designed to satisfy these requirements.

CHAPTER 4

PADDI-2: Architectural Design

In this chapter, a single-chip multiprocessor architecture called PADDI-2, which targets the rapid prototyping of high throughput DSP applications, is described. First, we summarize the architectural requirements for handling the target DSP algorithms described in Chapter 2. In the following sections, the control structure, processing element architecture, communication network design, memory and I/O architecture of the PADDI-2 architecture are presented, key design choices are explained and the various techniques used to achieve high performance are discussed. Alternative architectures are suggested and discussed. Here, we focus on the high level architectural ideas of PADDI-2. In the next chapter, the detailed hardware design of the architecture will be presented.

4.1 Architectural Requirements for Rapid Prototyping

The target architecture must be able to provide the high computation throughput of the order of 10 to 100 GOPS required by high performance DSP algorithms. Moreover, the architecture must be able provide high memory bandwidth and I/O bandwidth (hundreds of MBytes per second) in order to sustain the high computation throughput. In addition, the architecture must be generic and flexible enough to handle a wide range of algorithms, especially those with heterogeneous data communications. Emphasis is put on ease of programming and compilation in order to achieve rapid prototyping. Equally important is the scalability and modularity of the architecture. The architecture must be easily extended to tackle more complex problems, and to keep pace with the rapid advances in technology without radical redesign. Field-programmability is desired to reduce turn-around time and cut costs.

4.2 PADDI-2 Control Structure

4.2.1 Comparisons

The control structure is perhaps the most important aspect of a multiprocessor architecture. Not only does it dictate the programmability of the architecture, it also has tremendous effects on the efficiency of the system by handling the interactions between various components of the architecture. In this section, three canonical control structures, namely, centralized, distributed, and hierarchical, as illustrated in Figure 4-1, are discussed and compared.

Centralized Control

The centralized control structure uses a single global controller to control all the execution units (Figure 4-1(a)) and is often used for tightly coupled multiprocessor systems, e.g., the processor array for image processing in [38, 111]. The key advantage of this control strategy is its conceptual simplicity, which eases compiler development. Centralized control simplifies implementation of global program jumps, e.g., task swapping and context switches. By putting all resources under the control of a single program, dynamic resource sharing can be easily managed and implemented to enhance the efficiency of the architecture.

The centralized control structure works very well when all the processing elements of the

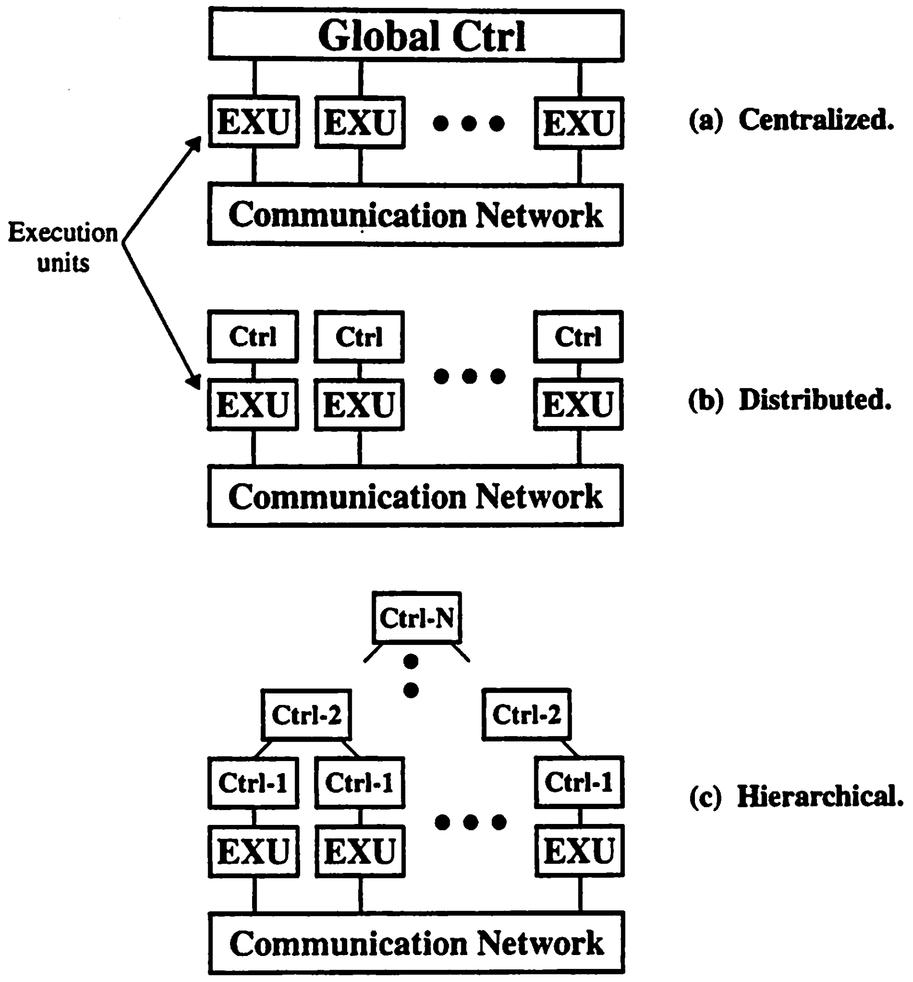


Figure 4-1: Comparison of 3 Control Structures.

system execute an identical or similar instruction thread. However, many loosely-coupled concurrently operating units are often used in the processing of complex DSP algorithms. In such a case, the centralized control can become very inefficient. For example, suppose there are two execution units, each controlled by a local controller, and N1 and N2 instructions are needed, respectively. In the case where the control flow of the two local controllers are independent of each other, up to N1 x N2 instructions are required if we are to use a single global controller for both execution units. Moreover the global controller must also examine two sets of status from the execution units and perform multi-way branches. It is clear that the complexity of the global controller is more than the total complexity of the two local controllers, and the same argument can be extended to higher number of execution units. Interestingly, this is very similar to the common observation that it is often possible to decompose a complex finite-state machine (FSM) into several simple interacting FSM's to save hardware.

Another inefficiency of the centralized controller is the long branch latency. Compared to a local controller, a centralized controller has to be located further away from the execution units both in physical distance on the chip and in hierarchy. This incurs higher branch penalty because it takes longer before the controller *receives* the status of the execution units and changes the program flow. For example, the use of an off-chip global controller in the first generation PADDI architecture results in multiple-cycle branch latency. The extra branch penalty can be reduced by substituting branch instructions with conditionally executed instructions [102], or by the use of special branch handling hardware such as the loop controller in [34]. However, in general, these features cannot be applied to all program jumps and it is not clear if a compiler can make effective use of them.

Even the application of the centralized control strategy to a moderate number of processors would require a very long instruction word, of the order of hundreds of bits [59]. This results in tremendous instruction bandwidth requirements, high costs in silicon area for instruction routing, and high I/O bandwidth requirements if the instructions have to come from off-chip sources. The use of local instruction buffers or decoders has been proposed to relieve the problem of instruction bandwidth bottleneck [27, 34]. However, these approaches are likely to add to the branch penalty further by introducing yet another level of control hierarchy. Except for single-instruction stream multiple-data stream (SIMD) systems, a purely centralized control structure is not practical for large systems due to technology constraints. Consequently, systems adopting the centralized control structure suffer from limited scalability.

Distributed Control

In a distributed control structure, each execution unit is equipped with a local controller (Figure 4-1(b)). The most important advantage of the distributed approach over its centralized counterpart is the excellent scalability of the system. More processor elements can be easily and seamlessly integrated into the system to tackle more complex problems. The scalability issue is going to be more and more crucial as technologies continue to advance and millions of transistors can be expected on a single die, and billions with wafer-scale system integration [99].

With the distributed control strategy, clusters of processing elements can work together on the same task or independently take on different tasks. Without the artificial boundary imposed by the control structure, complex system can be partitioned into sub-systems in the most natural manner. This facilitates the divide-and-conquer approach, promotes modularity and eases programming. Furthermore, by keeping the controller local, the instruction bandwidth bottleneck is eliminated and branch latency is shortened.

In a distributed environment, a local controller must somehow be able to co-ordinate its action with another local controller, e.g., by sending or receiving *control information* via the communication network. A shortcoming of distributed control is the overhead due to the exchange of the control information between the local controllers. An important kind of control information is the synchronization signal which directs the processing elements *when* a task can start. Correct system operation is ensured by synchronizing all the processing elements in the system to perform their tasks in the correct timing sequence. This form of control overhead is often called synchronization overhead. Synchronization issues can seriously impact system performance, especially in large-scale parallel systems, and therefore it is an area of very active research [24]. Another example of the control information is a *flag* sent by one processing element to another to control the latter to perform a specific task.

In general, programming using a distributed control structure is more tedious than using a global control structure because resources are managed by multiple independent controllers. Resource sharing and interactions between controllers must be carefully and explicitly handled by the programmer in order to achieve correct and efficient operation.

Another shortcoming of the distributed control structure is the limited capacity of the local program memory due to area constraints. However, it turns out this is not a very serious problem because by targeting high throughput DSP algorithms, the opportunity for multiplexing many operations onto the same processing element is limited, and therefore a simple controller with a small program memory is usually sufficient. Moreover it is always possible to reduce the demand on capacity by reassigning the operations to more processing elements at the expense of some inefficiencies. Further discussion on this subject will be given when the controller architecture of the PADDI-2 processing element is presented in Section 4.3.2.

Hierarchical Control

The hierarchical control structure is a compromise between the centralized approach at one end of the scale and the distributed approach at the other end. As implied by the name, this control structure is hierarchical: a group of controllers are controlled by a higher level controller, which in turn is controlled by yet another higher level controller. This control structure is very interesting in that it improves the system scalability of the centralized control and, at the same time, has less of the control overhead of the distributed approach. This control structure is particular attractive if the control hierarchy also matches the hierarchical partition of an algorithm.

The greatest drawback of the hierarchical control structure is its complexity. The multi-level control hierarchy makes automatic compilation of algorithms difficult. Moreover, the controllers on various levels are likely to have different requirements and hence different designs. The introduction of a variety of components into the architecture can complicate hardware design by degrading the regularity of the system.

If we focus the scope of our discussions to single-chip multiprocessor system, a control hierarchy with many levels is not practical. A particular interesting choice is a 2-level control structure that functions similarly to a typical host-coprocessor system. A very useful feature supported by such a control structure is task switching. While the low-level controllers are busy executing instructions stored locally, the top-level controller can prepare for a new task by writing new instructions to the program memories of the low-level controllers. Operation of the current task needs not be interrupted if instructions of the current task and those of the new task reside in different sections of the program memory. The top-level controller monitors the status information provided by the low-level controller to determine completion of the current task and then issues signals to the low-level controllers to switch to the new task. This feature is very useful especially for handling algorithms that consist of multiple distinct phases of computations. Besides task switching, this feature can be easily modified to accommodate exception handling and context switching as required in a multiprogramming environment.

Conclusions

From the above discussions, it is clear that the centralized control structure is a poor choice because of its limited scalability. Both the distributed and the hierarchical control structure offer attractive features. We finally decide to adopt the distributed approach for PADDI-2 mainly because the development of an efficient compiler, which is essential to rapid prototyping, is more straightforward for the distributed case. Moreover, the uniform control structure also enhances regularity of the architecture and therefore simplifies hardware design.

Nonetheless the capability to switch task without interruption, as described in the previous section, is a very desirable feature for the implementation of exception handling and context switches, etc.. Future generations of this architecture are likely to use control structures that support this feature.

As explained previously, programming on a distributed control structure can be difficult and

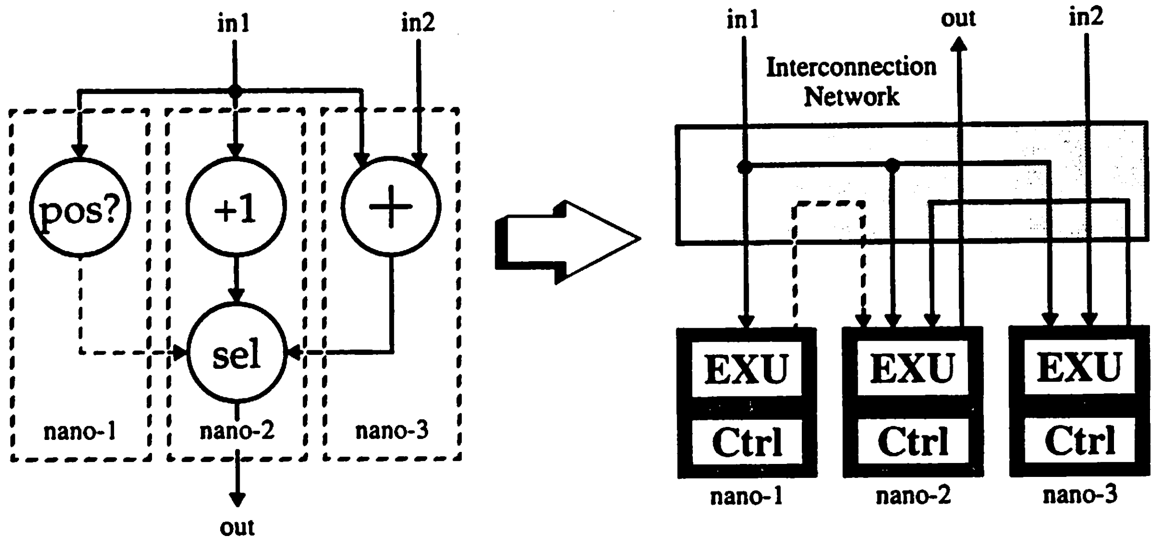


Figure 4-2: Direct Execution of Data Flow Graph using a network of Nanoprocessors.

tedious. In the next section, we will suggest a simple strategy to map algorithms onto distributed processing elements to ease the programming task.

4.2.2 Distributed Data-Driven Architecture

Basic Idea

In this section, the basic idea of the distributed data-driven PADDI-2 architecture is presented. Having decided on the distributed control structure, the key question is how to map algorithms onto processing elements controlled in a distributed manner while minimizing control overhead such as processor synchronization. The solution we propose is to directly execute a data flow graph by mapping nodes in the graph onto simple processing elements, called *nanoprocessors*, interconnected by a communication network, as illustrated in Figure 4-2.

A nanoprocessor is simply a hardware realization of a node or an encapsulation of several nodes. Similar to their data flow counterparts, nanoprocessors communicate with each other via communication channels (arcs in the graph). Execution of an operation (firing of a node) by a

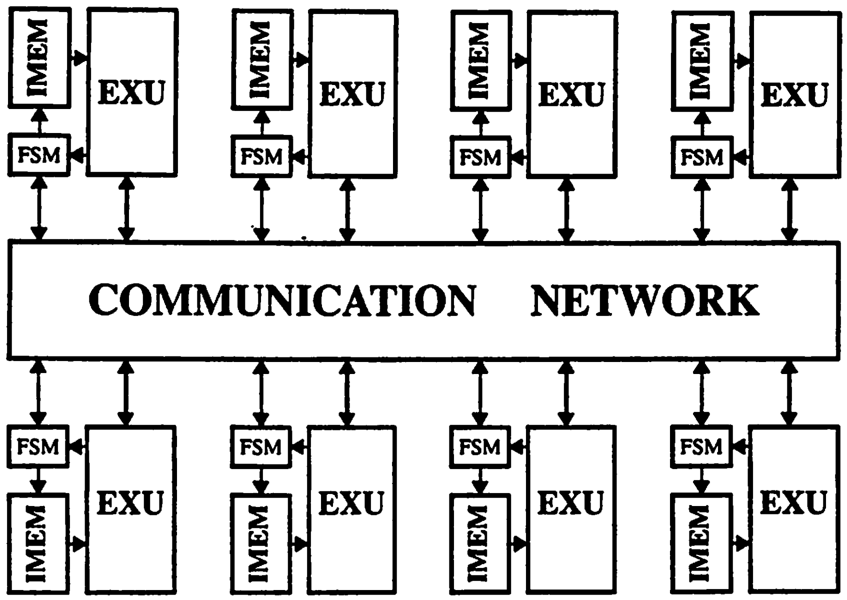


Figure 4-3: MIMD PADDI-2 Architecture Model.

nanoprocessor is *data-driven*, i.e., execution depends on the availability of the required operands. This implies that some form of handshake protocol is needed for any data transfer.

Each nanoprocessor independently executes instructions stored in its local program memory. The only way for a nanoprocessor to affect another nanoprocessor is by explicitly sending a message (token) to the latter via the communication network. This eliminates any *side-effects* the operation of one nanoprocessor may have on another, and is the key to ease of programming and good scalability. In essence, the architecture is very much like a multiple-instruction stream multiple-data stream (MIMD) message-passing multicomputer system, which is considered to be the most scalable multiprocessing system of all [13], except that very simple computers and communication network are used here. The MIMD PADDI-2 architecture model is shown in Figure 4-3.

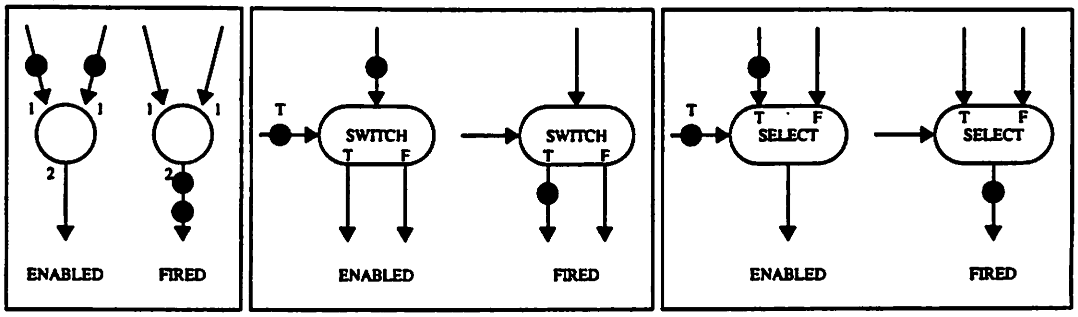
There are several important advantages for the architecture to support data flow semantics. It is widely accepted that a data flow graph is a natural, or perhaps the best, representation for signal processing algorithms [63]. Real signals are naturally represented by the data streams used implicitly by the data flow model. Moreover data flow graphs can easily expose very fine-grain parallelism in the DSP algorithms, which we need to exploit in order to achieve high performance. Finally since our architecture directly supports the semantics of data flow, the essential task of developing an efficient compiler for our architecture is greatly simplified.

The application of the data-driven approach to DSP is not entirely new. Many researchers have also discovered the elegance of this approach. In addition to the data-driven architectures discussed in Chapter 3, a highly modular and flexible DSP processor architecture consisting of multiple data-driven functional units was presented by Fellman in [34]. Another interesting recent example is the data flow processor for multi-standard video codec in [61] which is equipped with a flow graph programming tool. What distinguishes the PADDI-2 architecture most from these other architectures is the fine granularity of the data-driven processing elements and the use of a very flexible interconnection network.

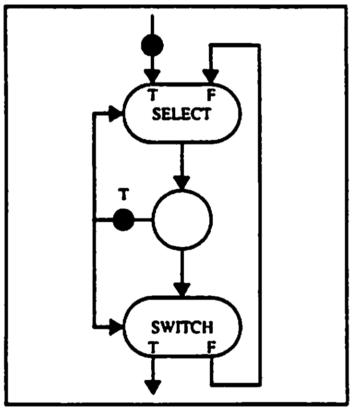
Static Data Flow

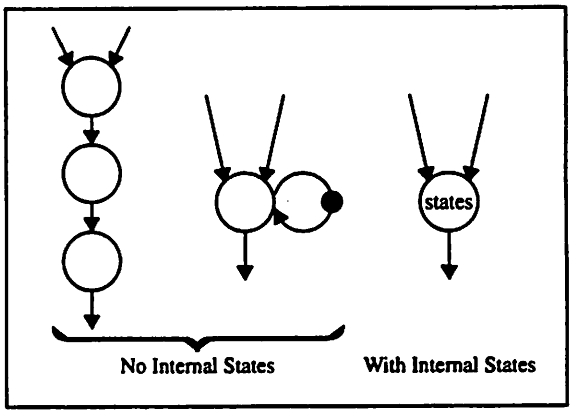
Before discussing the PADDI-2 architecture further, the static data flow model the architecture based on is first explained in this section. The discussions lead to the general definition of the basic structure of a *minimal nanoprocessor* which can be interconnected to implement any algorithms.

Among the many variations of data flow models [1], we adopt Dennis' classic static data flow model [28]. A data flow program is described by a directed graph where the nodes denote operations and the arcs denote data dependencies. Dynamic behavior due to data-dependent iterations and conditionals is handled by the SWITCH and SELECT operators. Figure 4-4(a) shows examples of the basic data-flow operators. The number next to an arc denotes the number of tokens consumed or produced on each firing and is omitted for clarity if it equals one. The nodes are enabled to fire when the required tokens are available on the input arcs. After firing, the input



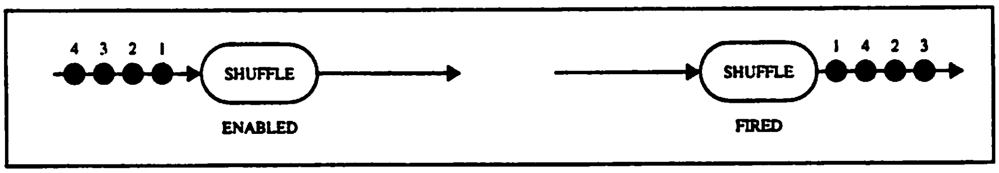
(a) Examples of data flow operators.





(b) Data flow graph with dynamic behavior: a while-loop.

(c) Three equivalent Data-flow graph representations.



(d) SHUFFLE operator.

Figure 4-4: Static Data Flow Model.

tokens are consumed and a number of result tokens are produced on the output arc. As an example, the SWITCH node steers the input token to either one of its output channels depending on the value of the boolean control token. In Figure 4-4(b), a while-loop implemented using the SWITCH and SELECT nodes is shown as an example of a data flow graph with dynamic behavior.

The arcs going in and out of the nodes represent physical connections called channels between the nanoprocessors. To simplify the implementation of channels, the First-In-First-Out (FIFO) schema is enforced on the arcs, i.e., the arcs are simply FIFO queues of bounded capacity and the tokens must be processed *in order.* If improperly programmed, the dynamic behavior and the bound capacity requirement can cause deadlocks. A detailed review of the properties of data flow graphs related to deadlocks can be found in [63].

A very important reason for the adoption of the static data flow model and the FIFO scheme is their simplicity. Simple hardware support such as first-in-first-out buffers and straight forward handshake circuits are sufficient for implementing the model. By contrast, in the more general dynamic data flow models, tokens can be processed *out of order* in order to expose more parallelism. A direct consequence of this generalization is that each token must be somehow *tagged* for proper identification and the hardware costs in storing, manipulating and matching of the tagged tokens can be overwhelming. In fact, this is one of the most serious problems that doomed the many general data flow machines proposed in the 70's and 80's [1].

Since data flow nodes communicate with each other exclusively using *streams* which are simply ordered sequences of tokens, all data structures have to be represented as such. Some examples of data structures commonly used in signal processing are the infinite stream (e.g., continuous speech) and the 2-dimensional matrix (e.g., an image block). More complex data structures can be represented using multiple streams. Stream manipulations such as merging or splitting can be handled by simple data flow nodes such as the SELECT node or SWITCH node, e.g., to partition a frame of pixels into multiple blocks for parallel processing.

We further divide streams into two kinds: data streams and control streams. Data streams carry data tokens, which generally represent certain signal characteristics, whereas control streams are used for control tokens, which contain boolean-type information, e.g., the select input to a SELECT data flow node. While the two are identical semantically, their implementations can differ significantly. For example, compared to the control streams, more bits are likely to be used for the communication channels carrying data streams.

A pure data-flow node does not have internal state. By allowing self-loops, a pure node can be extended to include internal states to represent an encapsulation of several nodes. Figure 4-4(c) shows the equivalence of the three representations. A cluster of nodes simply denotes a number of primitive operations executed in a specific order. Thus data-flow nodes can be implemented either spatially by a network of nanoprocessors, or temporally by executing a sequence of instructions on a single nanoprocessor. In the latter case, the nanoprocessor functions just like a traditional Von Neumann machine.

The FIFO schema dictates tokens be processed in order. To change the order, a SHUFFLE node as shown in Figure 4-4(d) can be used. In the simple case, the SHUFFLE node can be implemented using a single nanoprocessor by first storing the data in its internal buffer and reading them out in the desired order. In general, the data has to be first stored into memory and then read out in the desired order.

Any algorithm can be described by using only the three basic data flow nodes shown in Figure 4-4(a) [28]. Figure 4-5(a) shows the basic structures of three simple nanoprocessors, each of which can implement one of the three basic data-flow nodes. By coalescing the three structures into one, a minimal nanoprocessor is obtained which can implement all three basic data-flow nodes (Figure 4-5(b)). Therefore by interconnecting multiple such minimal nanoprocessors, any algorithms can be implemented. The implementation is, however, most efficient for algorithms that are computation intensive and require simple control, and these are precisely the characteristics of our

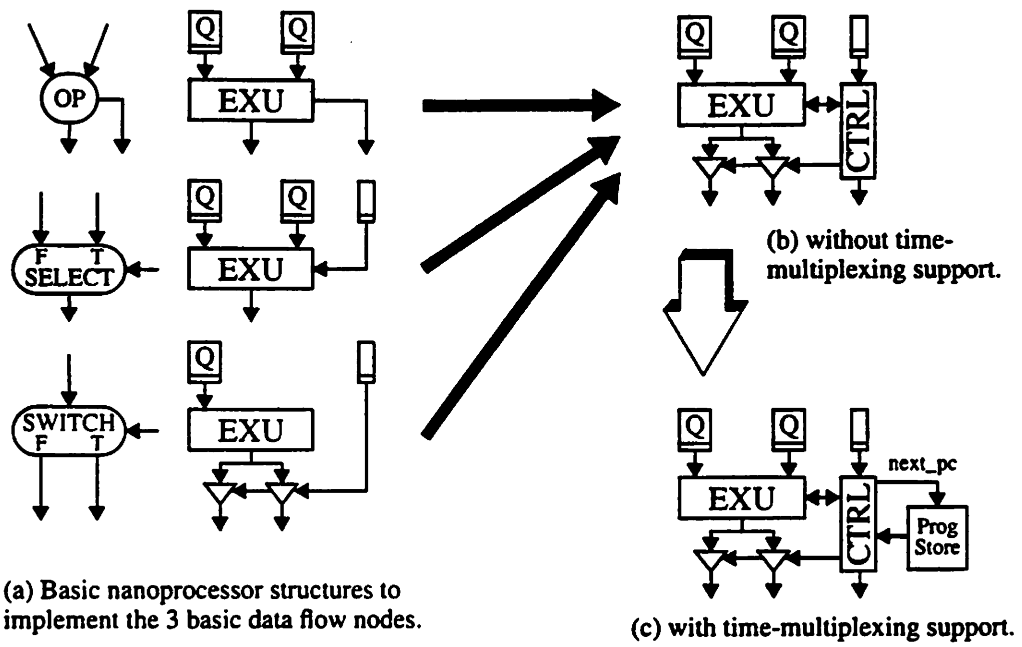

OP
Q
EXU
F T
SELECT
SWITCH
CTRL
next_pc
Prog Store
(a) Basic nanoprocessor structures to implement the 3 basic data flow nodes.
(b) without time-multiplexing support.
(c) with time-multiplexing support.

Figure 4-5: Minimal Nanoprocessor.

target applications. Sometimes efficiency can be gained by time multiplexing of several nodes onto a single nanoprocessor. To support time multiplexing, a small program store is added to the minimal nanoprocessor as shown in Figure 4-5(c). The minimal nanoprocessor is important because it defines the simplest nanoprocessor that can support the execution of data flow nodes.

Discussion

The advantages and disadvantages of the distributed data-driven control strategy used in PADDI-2 are summarized in this section. A very important advantage of the data-driven control strategy is its simplicity; the only requirements are the handshakes during communications and the checking of the availability of operands by the nanoprocessors. Synchronization of any number of nanoprocessors is handled automatically and gracefully by the data-driven principle of execution. The simplicity of this control strategy also simplifies the development of a compiler for the architecture.

As already mentioned at the beginning of this section, execution of a program in one nanoprocessor has no side-effect on the others. By eliminating a global controller and side-effects, good scalability and modularity of the architecture are ensured. Moreover the data-driven handshake protocol provides a uniform interface between *all* the components of the architecture which can be as simple as a single nanoprocessor or as complicated as a large sub-system. Consequently, several sub-systems can simply be *connected together* to build a complex system that is guaranteed to work as long as all the sub-systems are also functional. Scalability is excellent as nanoprocessors can be integrated easily and seamlessly into a system to tackle more complex problem.

Another important advantage is that the programming task is greatly simplified because programmers no longer need to worry about side-effects and synchronization is handled implicitly by the data-driven execution. Finally the control strategy keeps the controller of the nanoprocessor local. This relieves potential instruction bandwidth bottlenecks and allows the use of long and fully decoded instructions, resulting in a fast cycle time and a powerful instruction set. In addition, zero-cycle branch latency as well as multi-way branching can be achieved with very little speed penalty.

Unfortunately all the benefits mentioned above do not come for free. There is a significant cost in hardware to implement the handshake protocol and control functions for data-driven execution. For example, based on the prototype PADDI-2 chip to be presented in the next chapter, it is estimated that about 18% of a nanoprocessor in terms of area is devoted to these functions. Since the penalty is to a large extent independent of the width of the data path, obviously the area penalty is going to decrease if the implementation uses a wide data path. A big challenge in architectural design is to minimize the hardware overhead associated with the distributed data-driven control strategy while retaining most of its benefits. Some hardware techniques used to reduce such hardware overhead will be explained when the hardware design of a prototype PADDI-2 chip is described in the next chapter.

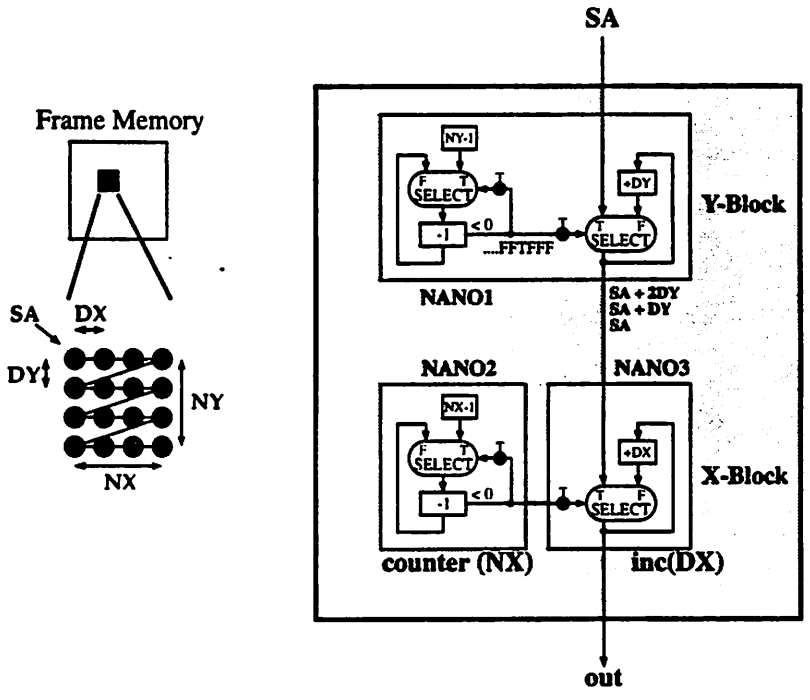


Figure 4-6: 2-dimensional Space Address Generator using 3 Nanoprocessors.

4.2.3 Address Generator Example

Figure 4-6 is a simple example that illustrates the mapping of data flow graph to several nanoprocessors. A two-dimensional space address generator [101] commonly used in video applications to read out a block of pixels from the memory is implemented using three nanoprocessors.

The data flow graph representing the address generator can be divided into 2 identical blocks: one for the y-direction and another for the x-direction. Each block consists of a periodic counter, counter(N), and an incrementer, inc(D). The parameters N and D correspond to the counter period and the increment, respectively. The X-counter (Y-counter) counts and generates a reset signal when the end of the line (block) is reached. The incrementor keeps incrementing the previous address by D until it receives a reset signal from the counter, at which point it resets the address to the input it receives and repeats the process again. To provide a throughput of one address per cycle, the counter (NANO2) and inc (NANO3) for the x-direction have to operate every cycle, but the corresponding blocks for the y-direction can be run at a much lower rate and therefore they can be implemented using a single nanoprocessor (NANO1).

Using only three nanoprocessors, a programmable and generic 2-dimensional space address generator is implemented. Both the dimensions of the addressed block as well as the address increment can be varied as parameters. Moreover, a new address block can be generated simply by changing the input starting address (SA).

4.3 Processing Element Architecture

4.3.1 Homogeneous vs. Heterogeneous

In this section the trade-offs between homogeneous processing elements and heterogeneous processing elements are examined. In a DSP algorithm, a variety of operations such as additions and multiplications are typically needed. By allowing heterogenous processing elements, they can be optimized by targeting one or a specific set of operations, e.g., a fast parallel multiplier can be included in processing elements that handle the multiplication operations. The solution is especially efficient if the numbers of the different types of processing elements match the exact requirements of the algorithms. In this case, both hardware saving as well as performance improvement can be achieved.

Unfortunately many of these advantages are easily offset by the disadvantages. In general the relative proportions of the different operation types vary considerably over a wide range of DSP algorithms and therefore, more often than not, processing elements dedicated to a particular operation will be under-utilized. For example, considering the three DSP systems described in Chapter 2, a processing element equipped with a parallel multiplier can handle the many filtering operations in the HDTV digital camera very well but the multipliers will be completely idle in the other two applications!

In view of the rapid advances in VLSI process technology, millions of transistors can be expected on a small die by the turn of the century and therefore the saving in silicon area, achieved by using dedicated functions compared to general regular programmable functions, is diminishing. Moreover the regular structure obtained by using homogeneous processing elements is much easier to design and test than irregular structures. The ease of designing structured hardware also implies that the designer can afford more time in optimizing the various aspects of the design, such as speed, area and power consumption.

Furthermore, it is more efficient to dedicate hardware at run time, rather than at design time, because program changes can be made much faster than hardware changes. Another problem is the difficulty in designing a flexible high bandwidth communication network interconnecting heterogeneous processing elements. Last but not least, heterogeneous architectures are nightmares for compiler designers because it is much more difficult to conceptualize and capture the architectural model into the compilation algorithms. Many more issues, such as the hardware mapping and the physical placement of the various kinds of processing elements, have to be considered.

To conclude, unless the application scope is well defined and fairly specialized, there is much to lose but little to gain by using heterogeneous processing elements and therefore we decided to use homogeneous nanoprocessors for the PADDI-2 architecture. The next step is to determine the appropriate complexity of the nanoprocessors. The design considerations will be discussed in the next section.

4.3.2 Processing Element Complexity

To satisfy the high throughput requirement, our architecture must be able to take advantage of the parallelism inherent in most of the target algorithms, down to a very fine-grain level. Before we discuss the *right* granularity for a nanoprocessor, we would first compare coarse-grain parallelism with fine-grain parallelism and discuss the features of the processing elements needed to exploit each kind of parallelism.

Fine-grain vs. Coarse-grain Parallelism

Two hypothetical data flow graphs, DFG1 and DFG2, are shown in Figure 4-7 to illustrate the differences between fine-grain parallelism and coarse-grain parallelism. Both data flow graphs implement the same algorithm consisting of N operations. DFG1 contains very fine-grain

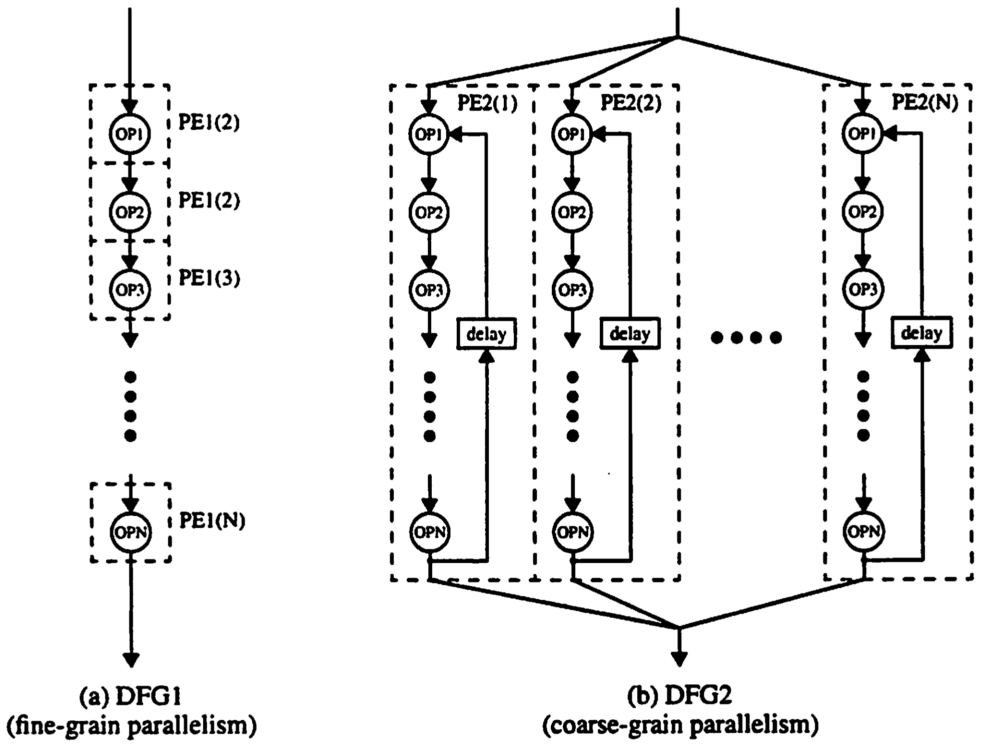


Figure 4-7: Examples of data flow graphs with fine-grain and coarse-grain parallelisms.

parallelism because the graph can be pipelined down to a single operation. Suppose a processor can perform one operation per cycle. Using N processors (PE1's), one for each operation, a computation throughput of N operations per cycle is achieved.

On the other hand, due to the single delay in the recursive loops in DFG2, the smallest unit of parallelism one can exploit profitably is N operations. In other words, DFG2 contains coarse-grain parallelism. However, if all N recursions can be processed independently as shown, again a computation throughput of N operations per cycle can be achieved by using N processors (PE2's), one for each loop.

Despite the identical throughput performances, the architectural requirements of PE1 exploiting fine-grain parallelism are very different from those of PE2 dealing with coarse-grain parallelism. The control unit of PE1 can be minimal since it is dedicated to a single operation whereas PE2 must have fairly a complex control unit to execute the N operations sequentially. Moreover, the execution unit of PE2 must be powerful enough to perform all N operations, which may be of a large variety. At first glance, it seems that a similar requirement is imposed on the execution unit of PE1 if heterogeneous processing elements are not allowed. However, if the complex operations can be decomposed into simple operations and processed in a pipelined fashion by using several processing elements, the same throughput can still be achieved by using PE1's equipped with a very simple execution unit. Although more PE1's are needed, the complexity of PE1 can be much less than PE2 and the overall hardware saving can be tremendous especially if the majority of the operations are simple.

In short, both the control unit and the execution unit of PE1 can be simpler than those of PE2 without performance degradation. We can therefore conclude from this example that a fine-grain processing element is more appropriate for exploiting fine-grain parallelism, and vice versa.

Simple Nanoprocessor

In multiprocessing, the choice of the complexity of the processing elements has a big impact on the efficiency of the architecture. If the processing element is overly simple, more of them are needed to achieve a certain throughput and more pressure is put on the interconnection network design. On the other hand, if the processing element is too complex, a good portion of it may be under-utilized, wasting valuable silicon area and often resulting in a slower processor because of the overhead. Obviously, the ideal processing element depends heavily on the particular application at hand. One compromising solution is to use heterogeneous processing elements. However, as pointed out in Section 4.3.1, the problems associated with the heterogeneous resources are likely to more than offset the advantages.

To achieve high performance, our architecture must be able to take advantage of the abundance of fine-grain parallelism inherent in the target DSP algorithms. As we have discussed in the previous section, fine-grain processing elements are more suited for exploiting fine-grain parallelism. As an example, let us consider an algorithm consisting of roughly the same number of multiply operations and add operations. The following assumptions are made:

- The size of a processing element with an adder is 1 unit area.
- The size of a processing element with a parallel multiplier and an adder is 2 area units.
- A multiplication operation can be performed using four adders in a pipelined manner, achieving the same throughput.

Suppose there are N multiply and N add operations. A total of 2(N + N) = 4N area units is needed using processing elements with parallel multipliers, versus (4N + N) = 5N area units when using processing elements with adders only. This example shows that even for algorithms with a large proportion of multiply operations, a system built from processing elements with parallel multipliers is only slightly more efficient, in terms of area, than a system consisting of simple processing elements. However, even in algorithms dominated by multiply operations, many miscellaneous operations such as address generation, data routing and simple arithmetic operations are still needed. Moreover, many algorithms require none or a very small number of multiply operations. In these cases, the simple processing element will surpass its more complicated counterpart in terms of area efficiency because the built-in parallel multipliers are mostly wasted. Therefore, in general, systems using simple processing elements as basic building blocks are likely to be more area-efficient.

We thus believe that more efficient use of hardware and flexibility are achieved by defining a simple homogeneous nanoprocessor architecture. More complex functions can then be performed by configuring these basic units to work in parallel. In addition, by keeping the nanoprocessor simple, both cycle time and layout density can be improved.

4.3.3 Instruction Set Architecture

The instruction set architecture of PADDI-2 is presented in this section. The availability of a local instruction store has a strong influence on the design of the instruction set architecture. Since bandwidth for instruction fetching is no longer an issue with a local store, a long instruction word can be used instead of multi-word instructions, which hurt performance. Moreover, a horizontally decoded instruction format can also improve speed by simplifying instruction decoding. Thanks to the small size of the instruction store, the decoded instruction format can also save area by eliminating some decode logic.

Our design philosophy is to follow the very popular idea based on Reduced Instruction Set Computer (RISC). Only simple instructions that take one cycle to execute are provided. Complex instructions are excluded because they either demand expensive hardware support in the execution unit, or require multiple cycles to execute, degrading the performance.

A rich and powerful instruction set for PADDI-2 is obtained using a 40-bit long instruction. The PADDI-2 instruction set includes all the basic arithmetic instructions and logic instructions.

Extension to multiple-word operands is handled by arithmetic instructions with carry. In addition, a Select instruction is provided to speed up the Maximum (MAX) and Minimum (MIN) operations frequently used in DSP algorithms such that the MAX/MIN operation can be executed by a Subtract-Select instruction pair in two cycles. An alternative is to add hardware to the execution unit to perform the MAX and MIN operations in a single cycle. Although the hardware cost is small (just a multiplexor), the modification is likely to impact the cycle time because the result of the comparison used to control the multiplexor is usually on the critical path.

Since multiplications arise very frequently in DSP algorithms, especially in filtering applications, performance can often be improved by providing some form of hardware support for multiplication. However, providing a multiplication instruction requires a parallel multiplier. As we have argued in the previous section, a complex nanoprocessor is less efficient in terms of hardware utilization and flexibility than a simple nanoprocessor. Moreover, in many applications such as filtering and low-level image processing, the coefficients are often set up in such a way that a simple execution unit with support for add-shift operation suffices. In situations where the execution of the multiplication operation is not time-critical, it can also be performed iteratively by simple add-and-shift operations. Chen has collected and analyzed the operator statistics of a suite of high throughput DSP algorithms in his thesis and found that, indeed, the parallel multiply operations, which are necessary for performance, account for less than 10% of the total operations [27].

As a compromise, we provide two instructions: Mstart and Mstep, to enhance multiplication throughput. The Mstart and Mstep instructions implement a single iteration of the sequential second-order booth multiplication algorithm such that a 16b x 16b multiplication can be implemented iteratively in 8 cycles using a processor.

In an effort to simplify the execution unit, only a 2-bit bi-directional shift operation is supported. However, in some applications shifting of more bits is useful, e.g., to multiplex two 8-

bit pixels onto an I/O bus to reduce bus bandwidth.

It is important to point out that even though only primitive instructions are provided for multiple-word arithmetic, MAX/MIN operations, multiplication and multiple-bit shifting, all these operations can be pipelined to achieve a throughput of one operation per cycle by using more nanoprocessors.

A PADDI-2 instruction consists of 5 fields, each of which is orthogonal and controls the action of a different hardware unit. For example, the next program counter field (NPC) is in charge of resolving conditional branches and the generation of the next program counter. A complete list of the PADDI-2 instructions, instruction encoding and their mnemonics can be found in Appendix A.2 and detailed descriptions of the PADDI-2 instructions will be given in Section 6.2 when the PADDI-2 assembler language is introduced.

The PADDI-2 instruction set is much more general and versatile than that of the first generation PADDI-1 [27], especially in the handling of logic operations, multiplications and branching. Nonetheless, some enhancements are worth considering such as support for sign-magnitude operations, multiple-bit shift, and higher order Booth multiplication step. These instructions will improve performance, but at the expense of hardware complexity.

4.4 Communication Network

In order to exploit the fine-grain parallelism available in many DSP algorithms, a large number of processing elements have to be used and interconnected by a high bandwidth communication network. The large cost in area dictated by the bandwidth requirement presents a formidable challenge in the design of the communication network for PADDI-2.

In an effort to reduce the area overhead, many array architectures provide rather restrictive interconnection networks such as a mesh network, which only allows neighborhood connections.

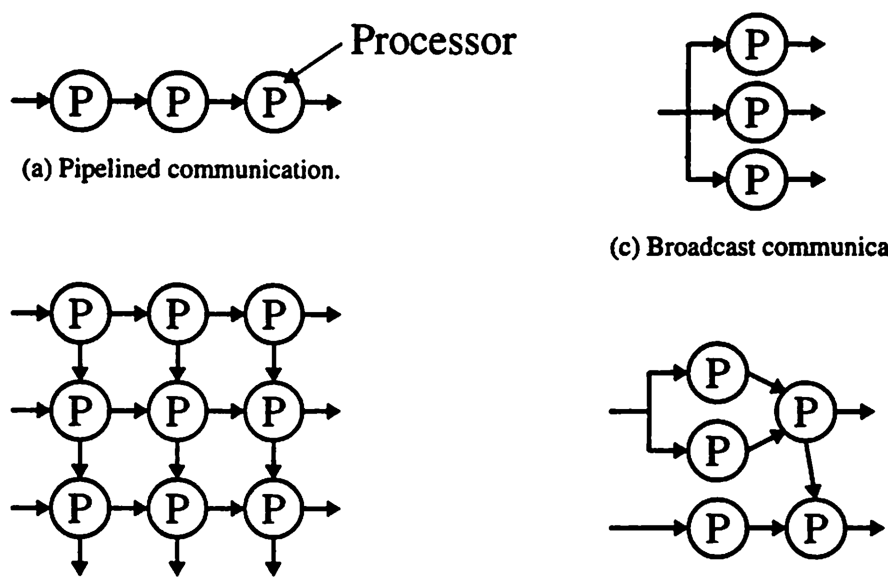
As we have already discussed in Section 3.3.2 where several array architectures were analyzed, the restrictive communication network limits the scope of the application. The lack of a flexible communication network also put a tremendous burden on the compiler to generate a good processor placement scheme.

Some array architectures implement non-local communications by routing the data successively through processing elements. A direct consequence of this approach is that the compilation process is complicated because the communication latency can vary a lot depending on the placement of the communicating processing elements. For algorithms with irregular communication patterns, this approach usually results in very long communication latency and poor processor usage. For systems with recursive structures, the long communication latency can lead to performance degradation. Furthermore, I/O to the interior of an array can be very inefficient.

Flexible Static Network

To overcome the above inefficiencies, our architecture provides a flexible high bandwidth communication network that can be reconfigured to match the communication patterns of a wide range of algorithms. Some examples of data communication patterns such as pipelined communication, mesh communication, broadcast communications, and etc., are shown in Figure 4-8. Sometimes, even though the intrinsic communication pattern of an algorithm is simple, the implementation may still demand a more flexible communication network. For example, if the linear system in Figure 4-8(a) involves multiple-word operations which are assigned to multiple processors, a mesh communication pattern as shown in Figure 4-8(b) is needed.

Motivated by the static interconnection used in connecting nodes in data flow graphs, we provide simple *static point-to-point* communications between nanoprocessors called channels, which are configured at system start-up and remain fixed at run-time. The point-to-point communication eliminates hardware overhead to support data routing. The run-time static design



(a) Pipelined communication.

(b) Mesh communication.

(c) Broadcast communication.

(d) General communication.

Figure 4-8: Examples of data communication patterns.

can also improve speed of the network because the large transistors typically used in implementing switches in the network do not have to switched every cycle as in a dynamic network. By keeping the communication network simple, valuable silicon area can be devoted to provide more channels to increase the communication bandwidth.

The lack of dynamic interconnection does not impose serious restrictions on the architecture. As we have pointed out in Section 4.2, nanoprocessors are usually dedicated to the processing of a few operations. Therefore channels are likely to be saturated with data and the opportunity for channel sharing is limited, especially for algorithms with high data rate. This inference is confirmed by a study on the interconnection patterns of a set of high throughput, high sampling rate DSP algorithms by Chen [27]. The results are shown in the bar chart in Figure 4-9, which

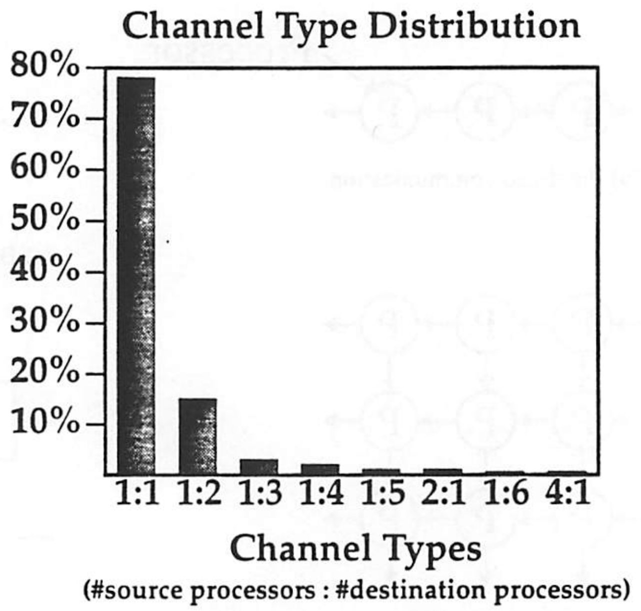
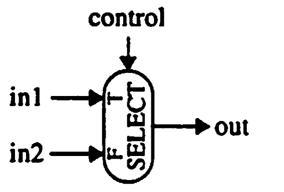


Figure 4-9: Channel type distribution for a set high throughput DSP algorithms [27].

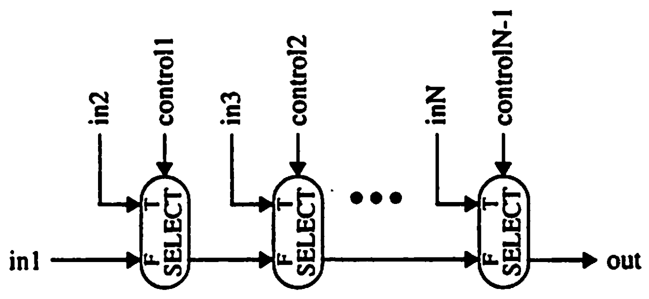
records the frequency of occurrences for different channel types. The channel type is represented by N:M, where N is the number of source processors and M is the number of destination processors sharing a channel. Any channel types other than the 1:1 type are shared channels. The results show that almost 80% of the channels are not shared.

Moreover, as will be shown in the next chapter, a nanoprocessor in our prototype PADDI-2 chip has a fan-in of 3 and a fan-out of 2 and therefore any channel type N:M with N less than 3 and M less than 2 can be handled by a single nanoprocessor. More complex channels which account for less than a few percents of the total can be emulated by programming nanoprocessors to implement the SWITCH and SELECT nodes. Figure 4-10 shows examples of several complex dynamic interconnect patterns realized by the SWITCH and SELECT operators.

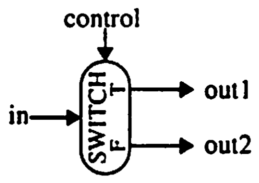
The run-time static network design is chosen because it optimizes the network design for the majority of cases where channels are not shared and suffers from some inefficiencies only in the



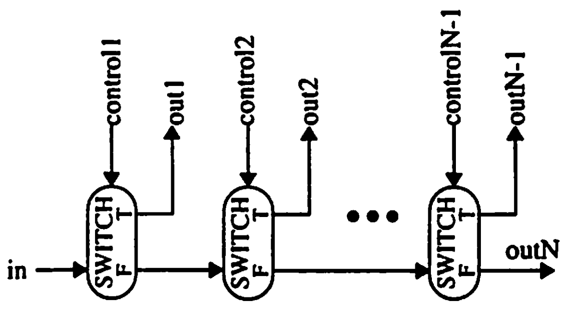
(a) 2-to-1 stream-merger.



(b) N-to-1 stream-merger.



(c) 1-to-2 stream-splitter.



(d) 1-to-N stream-splitter.

Figure 4-10: Examples of Dynamic Interconnection patterns implemented using SELECT and SWITCH operators.

infrequent cases where dynamic interconnections have to be emulated.

To enhance the efficiency of the communication network, data broadcasting, which is frequently used for data distribution when exploiting spatial concurrency, is supported. This feature also helps to eliminate the need for some dynamic interconnection patterns.

Hierarchical Network

To supply high bandwidth and flexible interconnect, many multiprocessor architectures, such as the Philips VSP and PADDI-1 described in Section 3.3.2, use a cross-bar network or variants of it. Although full connectivity is provided between any inputs and any outputs, the area complexity of a cross-bar network, which grows quadratically with the number of processing elements, is very large. For example, about as much as 18% and one-third of the die area are consumed by the cross-bar networks in the VSP chip and the PADDI-1 chip, respectively, even though these chips contain only a moderate number of processing elements. More importantly, the quadratic growth seriously limits the scalability of the network.

To reduce the area cost, we exploit the locality of communications inherent in the DSP algorithms. Even though hardware mapping and processor placement are very difficult problems for compilers, it is fair to assume that a reasonable compiler will be able to place communicating nanoprocessors physically close to each other most of the time. Consequently, local communications between nanoprocessors are far more frequent than distant communications.

To exploit this fact, a 2-level hierarchical communication network is proposed for PADDI-2 as a compromise between interconnect flexibility and area cost. The basic structure of the network design is shown in Figure 4-11. The level-1 network takes care of communications within a cluster containing several nanoprocessors and communications between the clusters are handled by the level-2 network. While the level-1 network can efficiently handle pipelined processors and local communications, the level-2 network can be used for generally irregular communications between nanoprocessor clusters. Compared to a cross-bar design, the hierarchical design is more area efficient with only a small penalty in interconnect flexibility. Moreover the design is highly scalable as more levels of network can be easily added to accommodate more nanoprocessors. Detailed descriptions of the VLSI design of the communication network will be given in the next chapter.

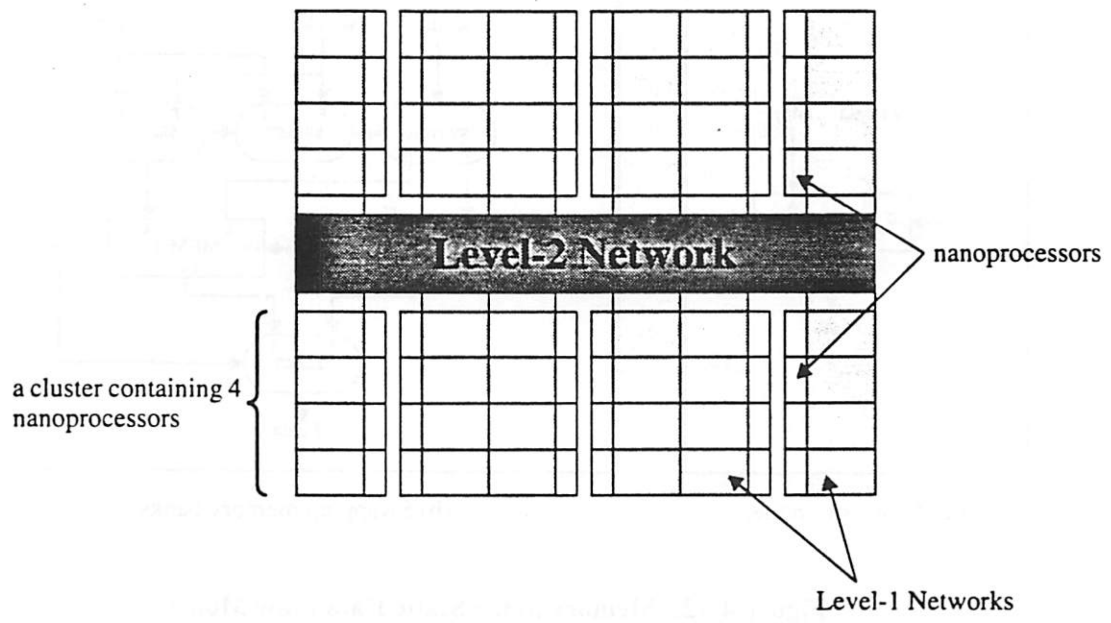
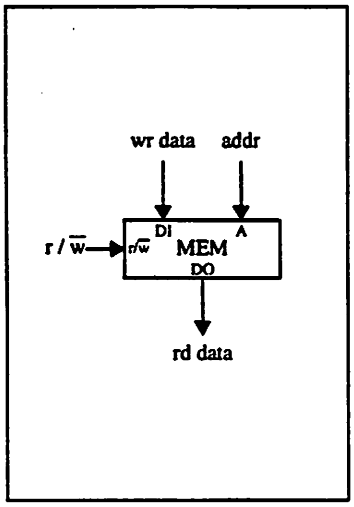


Figure 4-11: Basic Structure of the 2-level PADDI-2 Communication Network.

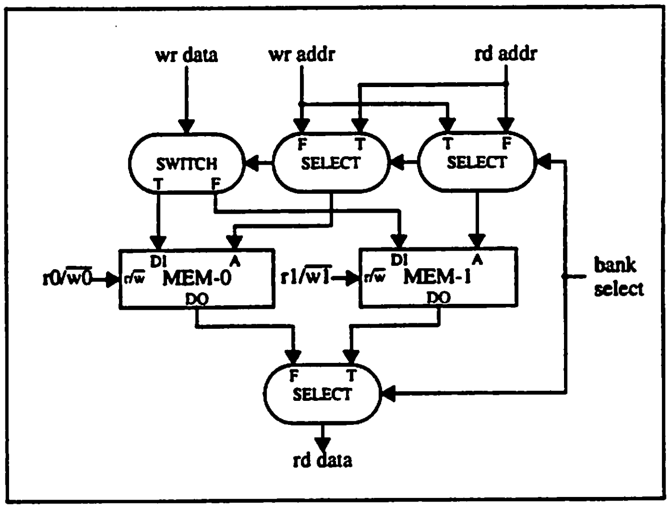
4.5 Memory Architecture

By allowing internal states in the data flow nodes, memories can be easily incorporated into the static data flow model and subsequently the PADDI-2 architecture. A memory node is just like any other data flow node except that it has a very large number of internal states. Hence a clean programming paradigm is preserved as every component of the architecture has a corresponding representation in the data flow domain.

As shown in Figure 4-12(a), a memory node consumes an address token and a control token specifying either a read or write operation. For a read operation, it generates an output token representing data stored at the location pointed to by the address token. For a write operation, it consumes an additional input data token and simply updates its internal states by writing the input data to the memory location addressed.



(a) A memory node.



(b) Swapping memory banks.

Figure 4-12: Memory in the Static Data Flow Model.

Memory nodes, together with the SWITCH and SELECT operators, can create many powerful shared memory structures. An example is the swapping memory banks shown in Figure 4-12(b). Depending on the bank select input, data can be written into one memory bank using one address sequence while data are being read from another memory bank using another address sequence. After all the data are written or read, the bank select input is toggled to switch the role of the two memory banks. This memory organization is often used in video processing, e.g., to buffer successive pixel frames.

Figure 4-13 shows the PADDI-2 architectural model incorporating memory elements (ME's) that are connected directly to the communication network and follow the same data-driven handshakes protocol as the nanoprocessors. To simplify the design, an ME is dedicated exclusively to memory read and write operations and therefore does not require an instruction memory. Any computations associated with memory access such as address generation are handled by the nanoprocessors.

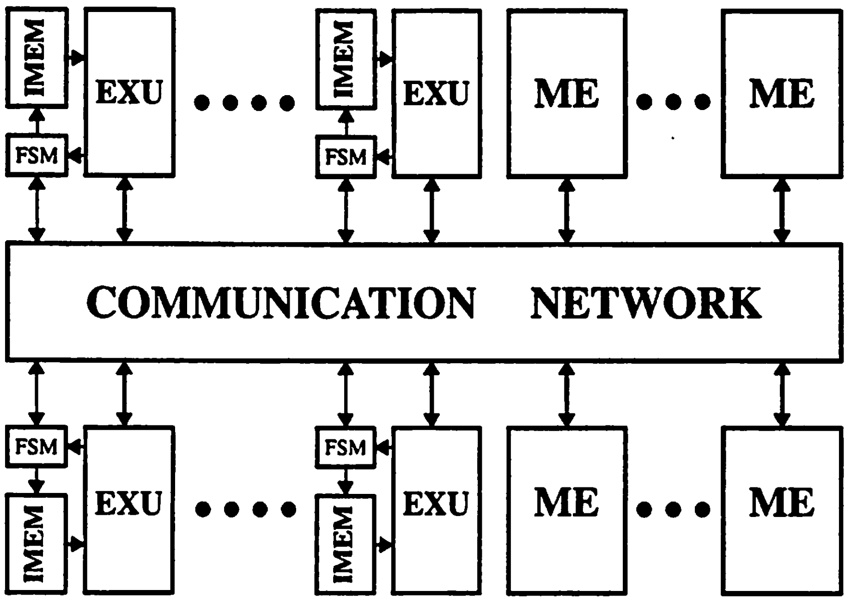


Figure 4-13: PADDI-2 Architecture Model with Memory Elements

4.6 I/O Architecture

To achieve high overall performance, a system must balance the computation power and the I/O bandwidth of the system to prevent either one from being the performance bottleneck. To sustain high throughput, the PADDI-2 architecture must be able to provide high I/O bandwidth to feed data into and retrieve results from the many fine-grain nanoprocessors on-chip. Besides performance, the understanding of the I/O requirement is very important because I/O bandwidth is very costly in terms of packaging and high speed I/O design is a very difficult problem especially on the board level [52].

The exact I/O bandwidth requirement varies depending on the system partitioning and the nature of the DSP algorithms. In general, the I/O requirement is more stringent for high data rate applications such as the high pixel rate HDTV digital color camera system described in Section 2.3. Chen in his thesis [27] examined the relation between the computation requirements and the I/O requirements of a set of high throughput DSP algorithms[1]. A geometric mean of about 4 MOPS/MByte/sec was reported. As will be discussed in the next chapter, a prototype PADDI-2 chip with 48 nanoprocessors can provide 2400 MOPS peak performance at the clock frequency of 50 MHz. According to Chen's findings, an I/O bandwidth of about 600 MByte/sec is needed. Using a 208-pin package, our PADDI-2 chip provides eight I/O buses of 16 bits width at the I/O rate of 50 MHz. This is equivalent to an I/O bandwidth of 800 MByte/sec, one-third more than the requirement obtained by Chen.

It is clear that as VLSI process technology advances, more nanoprocessors and therefore higher computation power will be available from a single silicon die. In order to prevent I/O bandwidth from limiting the performance, techniques on all levels such as communication-based hardware partition, multi-chip module packaging, and high-speed interchip signaling [69] must be applied.

In addition to the eight data I/O buses, a PADDI-2 chip also provides eight companion control I/O buses. The only difference between a control I/O bus and a data I/O bus is that the control bus is only 1 bit wide as opposed to 16 bits for the data bus. The control bus can be used to transfer boolean information between chips. As an example, a control bus can be connected to an off-chip static memory as read/write control.

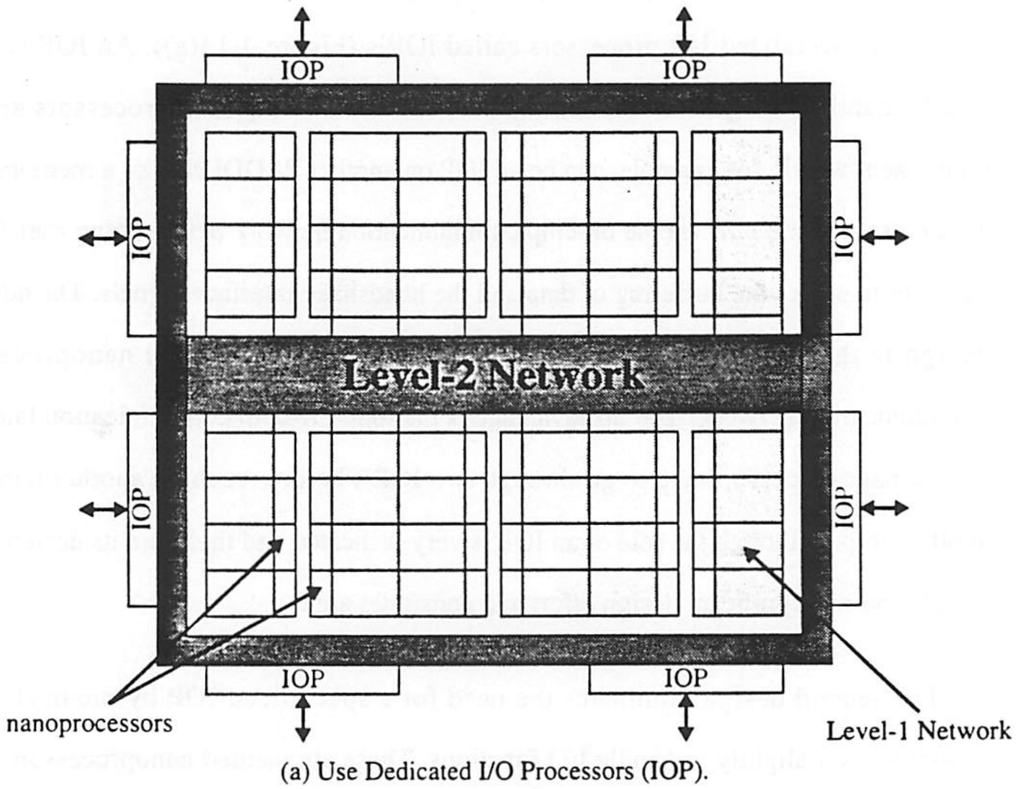
Because the data-driven principle of execution is used throughout the architecture, interconnection between all architectural components, such as nanoprocessors, memories or PADDI-2 chips, are all handled consistently by the same data-driven handshake protocol. The uniform interface greatly simplifies I/O design.

1. This is the same benchmark set used in the study of the interconnect characteristics discussed in Section 4.4.

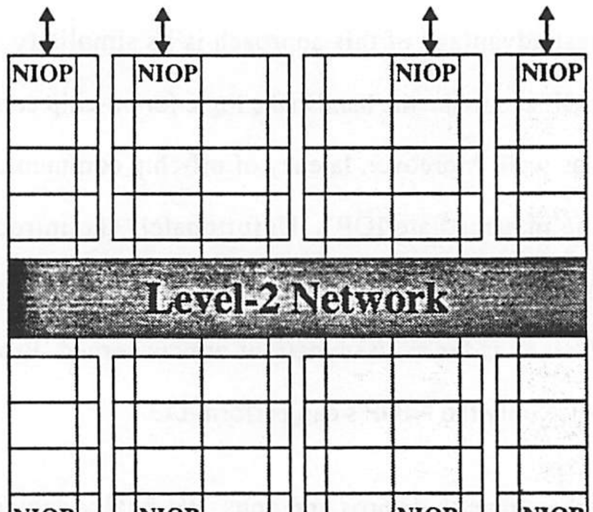
Two alternative I/O designs as depicted in Figure 4-14 are considered. The first design handles I/O using specialized I/O processors called IOP's (Figure 4-14(a)). An IOP functions as the middleman handling communications between the on-chip nanoprocessors and an off-chip component which, for example, can be an IOP on another PADDI chip or a memory module. Data are first sent to the IOP via the on-chip communication network before being transferred off-chip. The IOP manages the buffering of data and the handshake interface signals. The advantage of this design is that it is very general. All IOP's are accessible by all nanoprocessors via the communication network. The disadvantage is the long off-chip communication latency; data sent from a nanoprocessor have to go through two IOP's before reaching another nanoprocessor on another chip. Although the role of an IOP is very dedicated and therefore its design can be simple, it still requires significant design effort and consumes area.

The second design eliminates the need for a specialized IOP by modifying an ordinary nanoprocessor slightly to handle I/O functions. These augmented nanoprocessors called NIOP's are typically assigned to nanoprocessors located near the corners of the array as shown in Figure 4-14(b). The greatest advantage of this approach is its simplicity. Only minor modifications are needed for the NIOP's to make the handshake logic for on-chip communications work for off-chip communications as well. Moreover, latency of off-chip communications is reduced substantially by eliminating the intermediate IOP's. Unfortunately the introduction of NIOP degrades the homogeneity of the nanoprocessor array. Despite its resemblance to a regular nanoprocessor, an NIOP is very different as far as I/O functions are concerned. Processor placement is also more complicated because only the NIOP's can perform I/O.

Both approaches have their pros and cons. We finally decide to adopt the NIOP approach because it improves the off-chip communication latency and requires very little design effort.



(a) Use Dedicated I/O Processors (IOP).



(b) Use Augmented Nanoprocessors (NIOP).

Figure 4-14: PADDI-2 I/O Architectures.

CHAPTER 5

PADDI-2: VLSI Circuit Design

The architectural design of the PADDI-2 architecture was presented in Chapter 4. As a proof of concept and to verify the effectiveness of the architecture, a prototype PADDI-2 chip with 48 nanoprocessors was implemented. The hardware design of the prototype chip will be discussed in a top-down manner in this chapter.

5.1 48-Nanoprocessor Prototype Chip

A block diagram of the 48-nanoprocessor prototype chip is shown in Figure 5-1. The chip contains 48 16-bit nanoprocessor interconnected by a 2-level high bandwidth communication network. The nanoprocessors are organized in two arrays, each of which contains six clusters of four nanoprocessors. As described in Section 4.6, the eight nanoprocessors at the corners of the nanoprocessor array can also serve as I/O processors to handle off-chip communications with either other PADDI-2 chips or static RAM's without any external glue logic. The eight 16-bit I/O ports provide large bandwidth needed for sustaining the high computation throughput. Two ports can be parallelized to provide 32-bit I/O if necessary.

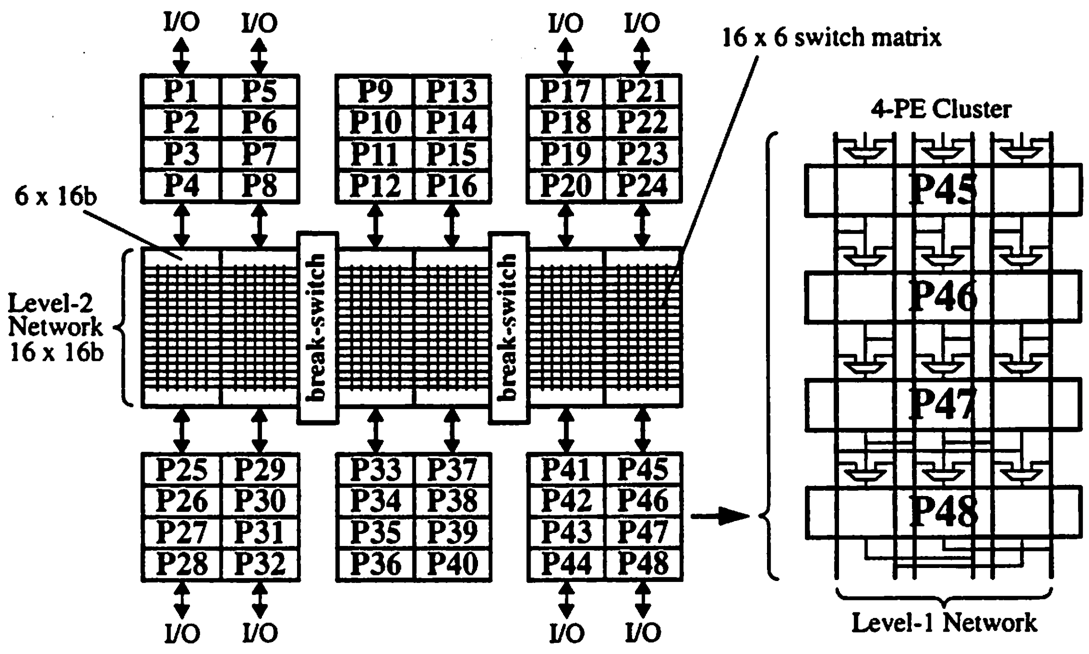

I/O
16 x 6 switch matrix
4-PE Cluster
6 x 16b
Level-2 Network 16 x 16b
break-switch
P1 P5 P2 P6 P3 P7 P4 P8
P9 P13 P10 P14 P11 P15 P12 P16
P17 P21 P18 P22 P19 P23 P20 P24
P25 P29 P26 P30 P27 P31 P28 P32
P33 P37 P34 P38 P35 P39 P36 P40
P41 P45 P42 P46 P43 P47 P44 P48
Level-1 Network

Figure 5-1: Block Diagram of 48-Nanoprocessor PADDI-2 Chip.

The chip has 208 pins including 170 utility pins and 38 supply pins. It contains 600K transistors and measures 12 mm by 12 mm and is packaged in a 208-pin ceramic pin-grid-array package. Running at 50 MHz, the prototype chip provides 2.4 GOPS peak performance and 800 MBytes per second I/O bandwidth. The micrograph of the prototype chip is shown in Figure 5-2 and a complete pin list can be found in Appendix A.1.

The chip was implemented using a 3-metal 1-μm CMOS process technology but only 2 layers of metal were used in the design in order to maintain compatibility with other process technologies. Being a scalable CMOS process, the design rules for the interconnect layers are very conservative. This has a big impact on the layout density of the chip because the design is mostly interconnect-limited. It is clear that utilizing the third layer metal not only can enhance the design of the communication network but also greatly increase the layout density of the chip. On the bright side, the process offers fast devices with effective channel lengths in the 0.8 μm range.

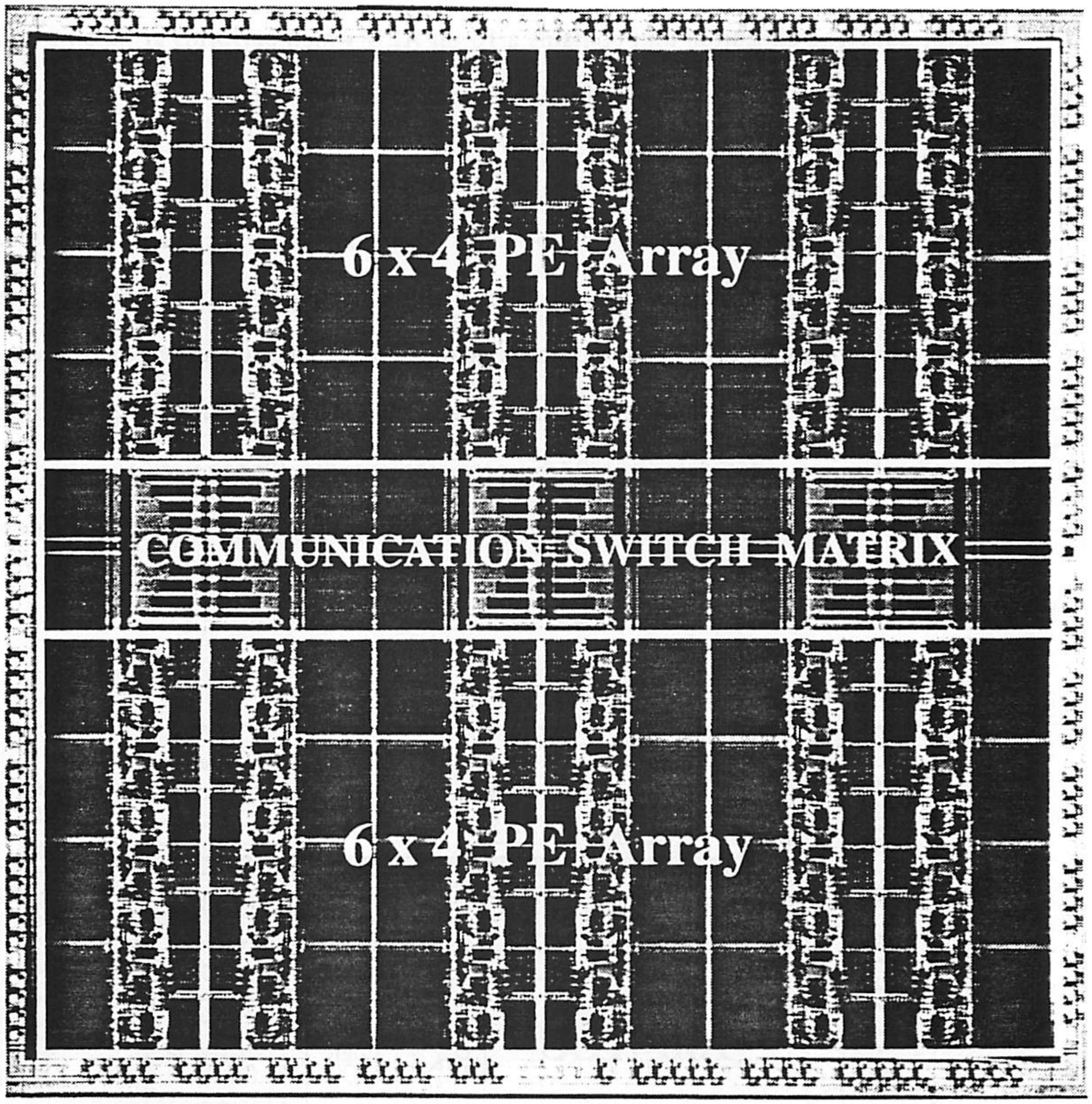


Figure 5-2: Micrograph of 48-Nanoprocessor PADDI-2 Chip.

5.2 Nanoprocessor Design

Figure 5-3 shows the block diagram of a nanoprocessor consisting of an execution unit and a control unit. The execution unit contains a 16-bit ALU which performs arithmetic and logic operations and one 2-word data buffer (DQ_i) for each of the three inputs.

Data path widths of 12-bit, 16-bit and 32-bit were considered. If the precision is not needed, a large width can potentially worsen performance by slowing down the cycle time and degrade hardware utilization. On the other hand, if the width is too small, a single operation has to be broken up into multiple operations, thus lowering throughput and/or increasing hardware demand. While a width of 12-bit is sufficient for most image and video algorithms, given the usual 8-bit representation of video data [43], 24-bit precision is typically required for quality audio applications [107]. We finally decided upon a 16-bit architecture as a compromise. Higher

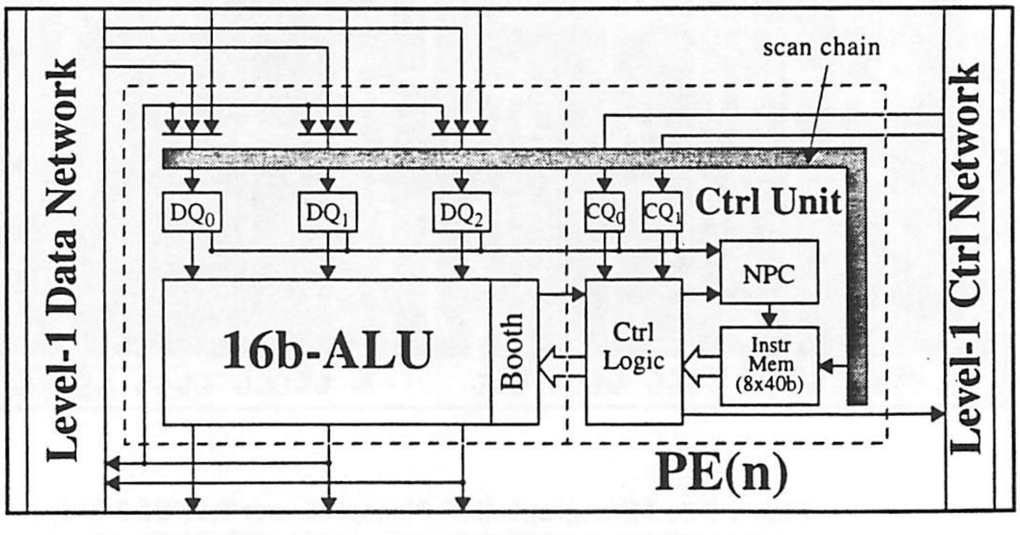


Figure 5-3: Block Diagram of PADDI-2 Nanoprocessor.

precision operations are performed using more nanoprocessors. Performance is not necessarily sacrificed if the multi-word operation can be pipelined and it is not part of a recursive bottleneck.

A detailed block diagram of the execution unit is shown in Figure 5-4. The ALU can execute a Conditional-Select instruction to speed up the MAXIMUM and MINIMUM operations often used in signal processing. Moreover, it is equipped with a Booth decoder to handle a modified Booth multiplication step instruction such that a 16-bit by 16-bit multiplication operation can be performed in eight cycles iteratively using one nanoprocessor or at 50 MHz rate by pipelining using eight nanoprocessors. The ALU is based on a fast and area-efficient carry-select adder with optimized block sizes. As shown in Figure 5-5, the block sizes of 3, 3, 4, 6 are chosen to minimize the total propagation delay by matching the propagation delay of the carry output of the individual block with the propagation delay of the global carry signal. The polarities of the inputs at each bit position are carefully chosen and dual complementary carry chains are used to avoid unnecessary signal inversion in the critical path to further enhance speed. The results of the critical path SPICE simulation of the carry select adder are shown in Figure 5-6. The worst case propagation delay of the carry output signal and the sum output signal are 5.3 ns and 6.3 ns, respectively.

The data buffer can be configured as a simple data queue or as a small random-access register file. When configured as a queue, the data buffer functions like a 2-word First-In-First-Out (FIFO) buffer. Data from the communication network are written to the tail of the queue and an instruction specifying the queue as its source operand will dequeue the data from the FIFO. An instruction can also use the FIFO as source operand but keep the data in the FIFO for reuse by the subsequent instruction. Hardware pointers automatically keep track of the status of the FIFO's. The enqueueing and dequeueing operations are stalled when the FIFO becomes full or empty, respectively. Inputs to the data buffers can be controlled by the scan chain to preset the contents of the registers, e.g., iteration bounds or filter coefficients, at system start-up. When used as a register file, each of the two registers in the buffer can be specified as source operand by an instruction.

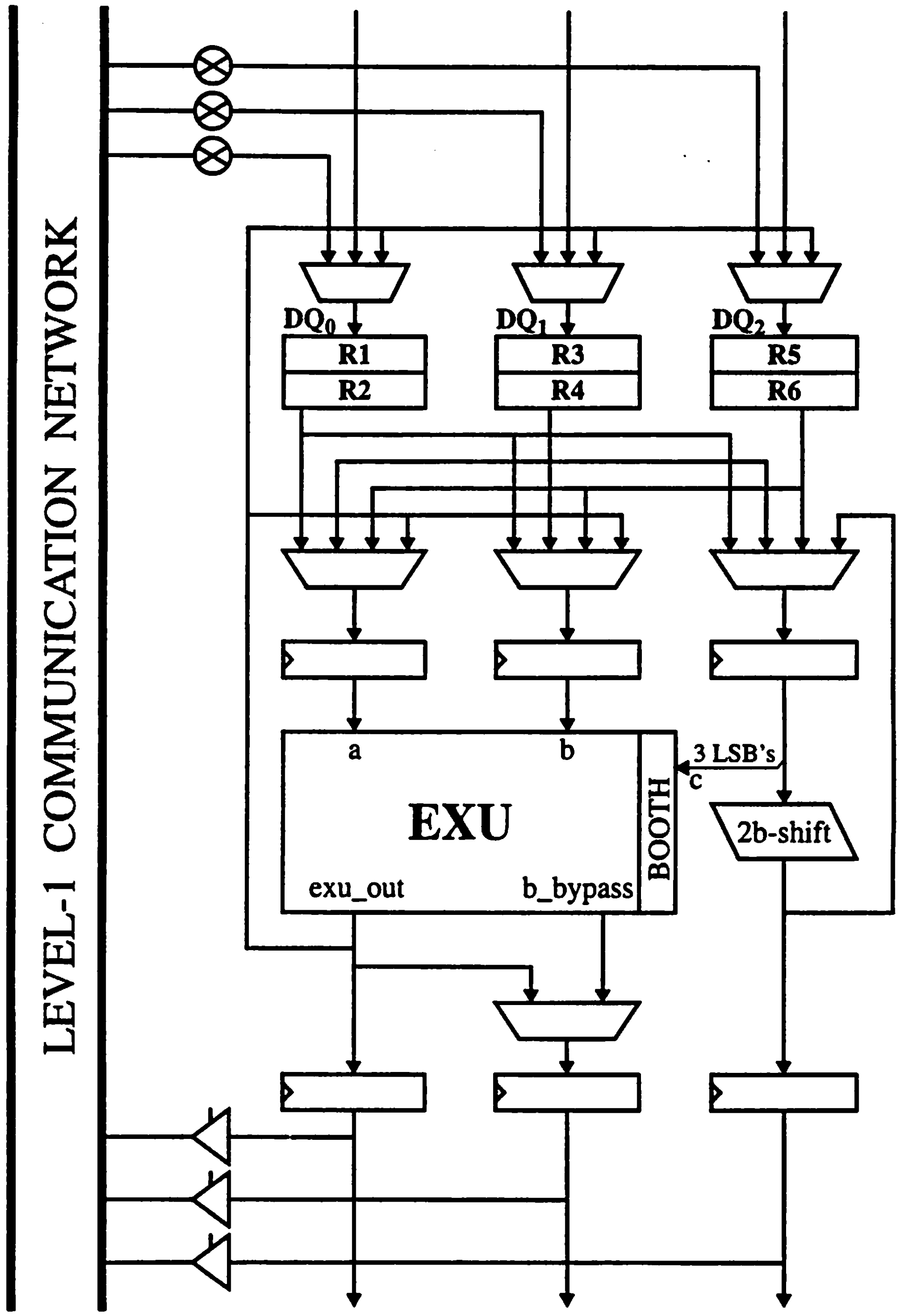


Figure 5-4: Block Diagram of Nanoprocessor Execution Unit.

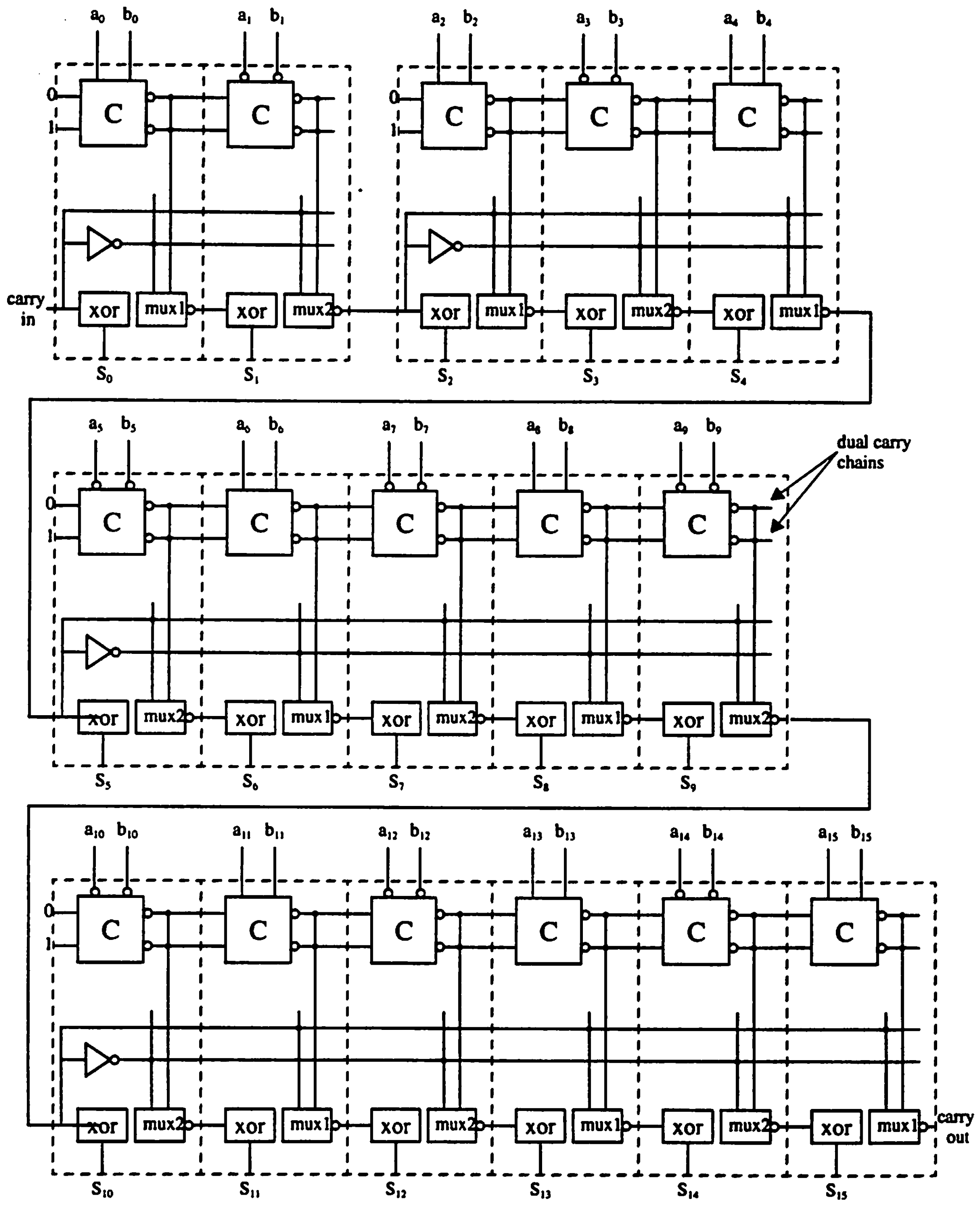


Figure 5-5: Block Diagram of 16-bit Carry-Select Adder.

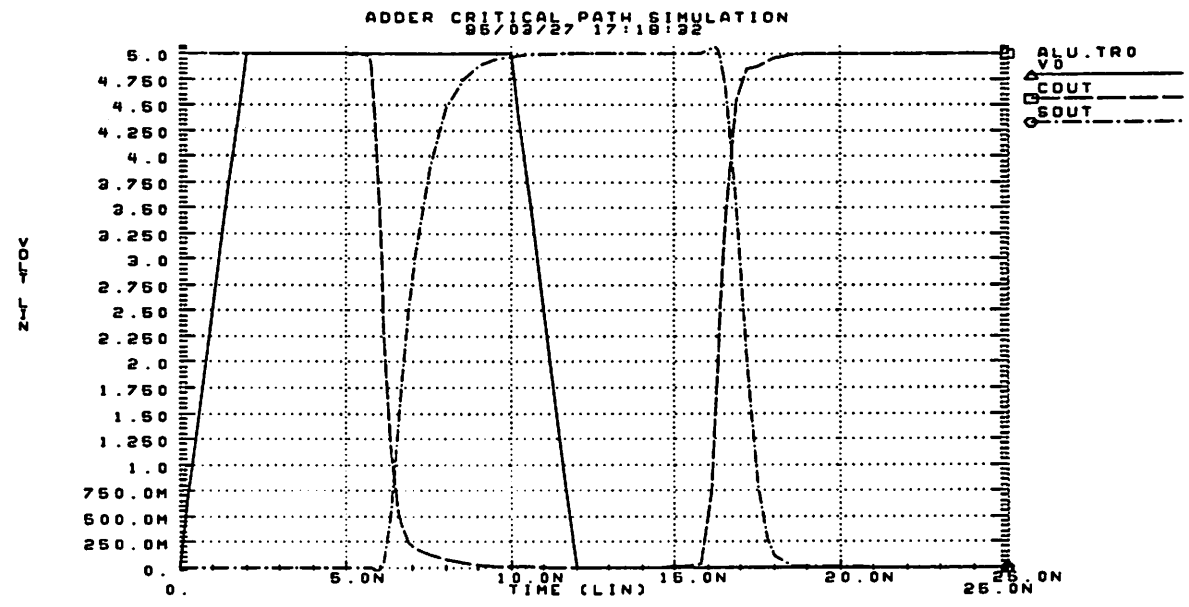


Figure 5-6: SPICE Critical Path Simulation of 16-bit Carry-Select Adder.

Three output connections to the Level-1 communication network are provided: two output connections are needed in order to implement the SWITCH node and the third connection is specifically provided to support pipelined implementation of the multiplication operation using multiple nanoprocessors as described in Figure 5-7. To facilitate neighboring communication and reduce network bandwidth demand, direct connections between neighbors are implemented without access to the Level-1 network.

The local controller contains a 40-bit by 8-instruction program store, a 3-bit program counter, two control queues for buffering the control streams and glue logic for decoding of control signals. Similar to the data buffers, the instruction store is initialized using the scan chain at start-up. The 40-bit horizontally decoded instruction simplifies the decode logic and improves cycle time. The program counter points to the next instruction to be fetched and can be modified depending on the status of the ALU or the control streams (CQ_i) to implement two-way branch with zero-cycle latency. The nanoprocessor is stalled if any of the required operands (data or control tokens) are

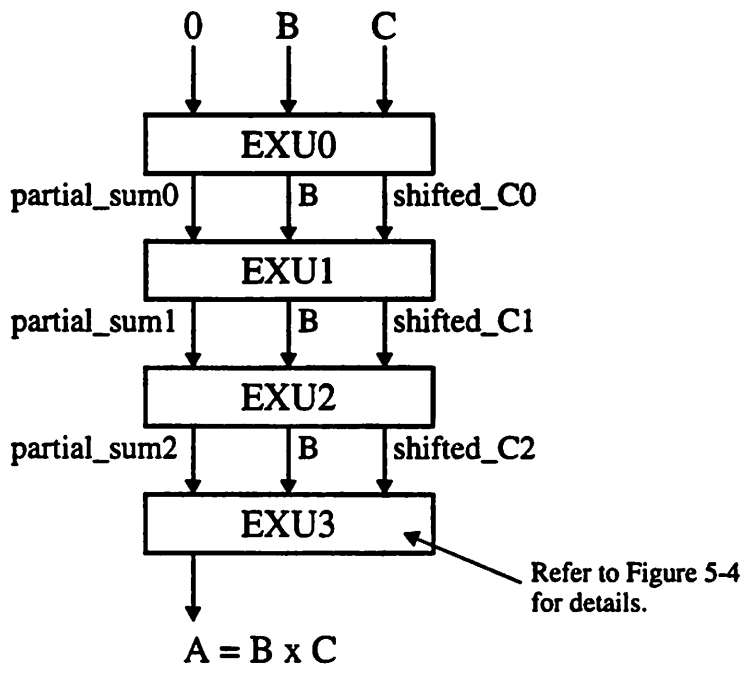


Figure 5-7: An example of pipelined implementation of multiplication.

not available or if any of the output channels are blocked.

A simple two-stage pipeline (Fetch and Execute) is used to speed up cycle time. The instruction and operands are fetched from the instruction store and data buffers respectively in the Fetch stage and the operands are processed by the ALU in the Execute stage. All instructions take one cycle to complete.

5.3 Communication Network

The hardware design of the communication network of the PADDI-2 prototype chip is presented in this section. As discussed in Section 4.4, the design is very challenging due to the high flexibility and high bandwidth requirements imposed by the architecture. To trade off flexibility in interconnection with area cost and cycle time, and to exploit the locality of communication, a design using a 2-level hierarchical network is adopted.

An overview of the basic structure of the two-level communication network has been shown in Figure 5-1 and more detailed description of the Level-1 communication network is given in Figure 5-8. The Level-1 network consisting of six 16-bit data buses and four 1-bit control buses is used for interconnecting 4 nanoprocessors within a nanoprocessor cluster. To save area, the network is laid out directly on top of the execution unit of the nanoprocessor as data path feed-throughs in second-layer metal. In the design of the prototype chip, the number of buses in the Level-1 network is constrained by the width of the execution unit. If a process technology with three routing layers was used, at least eight buses in the Level-1 network could be accommodated without expanding the width of the execution unit.

The basic structure of the 16 x 6 switch matrix in the Level-2 communication network is shown in Figure 5-9. The level-2 communication network consists of 16 data buses and 8 control buses and handles traffic between the 12 clusters. To take advantage of the fact that local communications are far more frequent than distant communications, break-switches as shown in Figure 5-1 can be opened to break up a bus (track) into a few buses of shorter lengths, hence increasing the effective number of buses in the network without area penalty. To prevent the delay in the communication network from dominating the cycle time, a channel can go through no more than one closed break-switch in the Level-2 network. This design is similar to the area-efficient interconnect matrix design in [107] except that the buses there are physically cut open.

Broadcasting is supported by the network to reduce demands on point-to-point communication channels and to facilitate exploitation of data parallelism inherent in many DSP algorithms. The broadcasting and the data-driven mechanism is efficiently supported by implementing the handshake signal (HS) associated with each bus as a dynamic WIRE-AND gate as depicted in Figure 5-10. Data transfer is completed only if HS remains high at the end of the

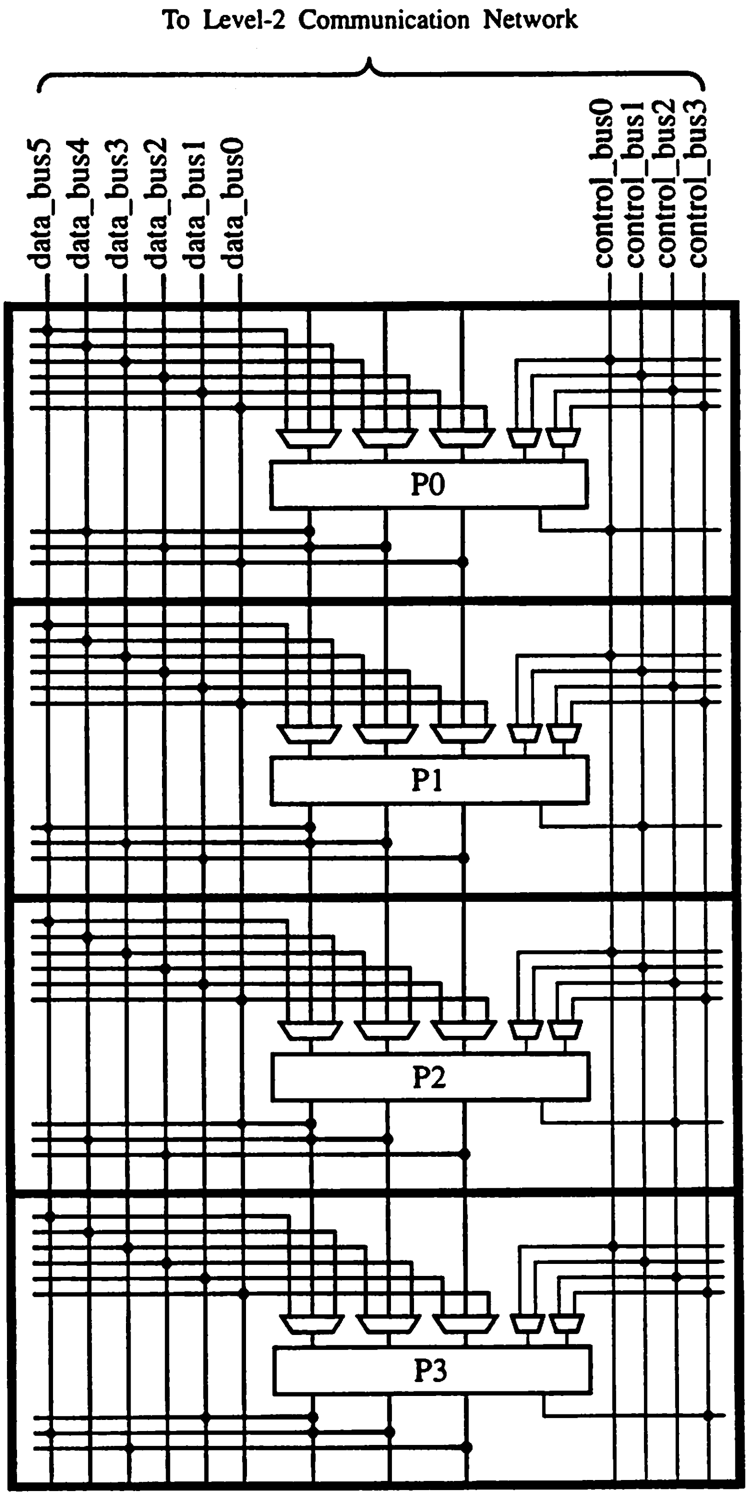


Figure 5-8: Level-1 Communication Network.

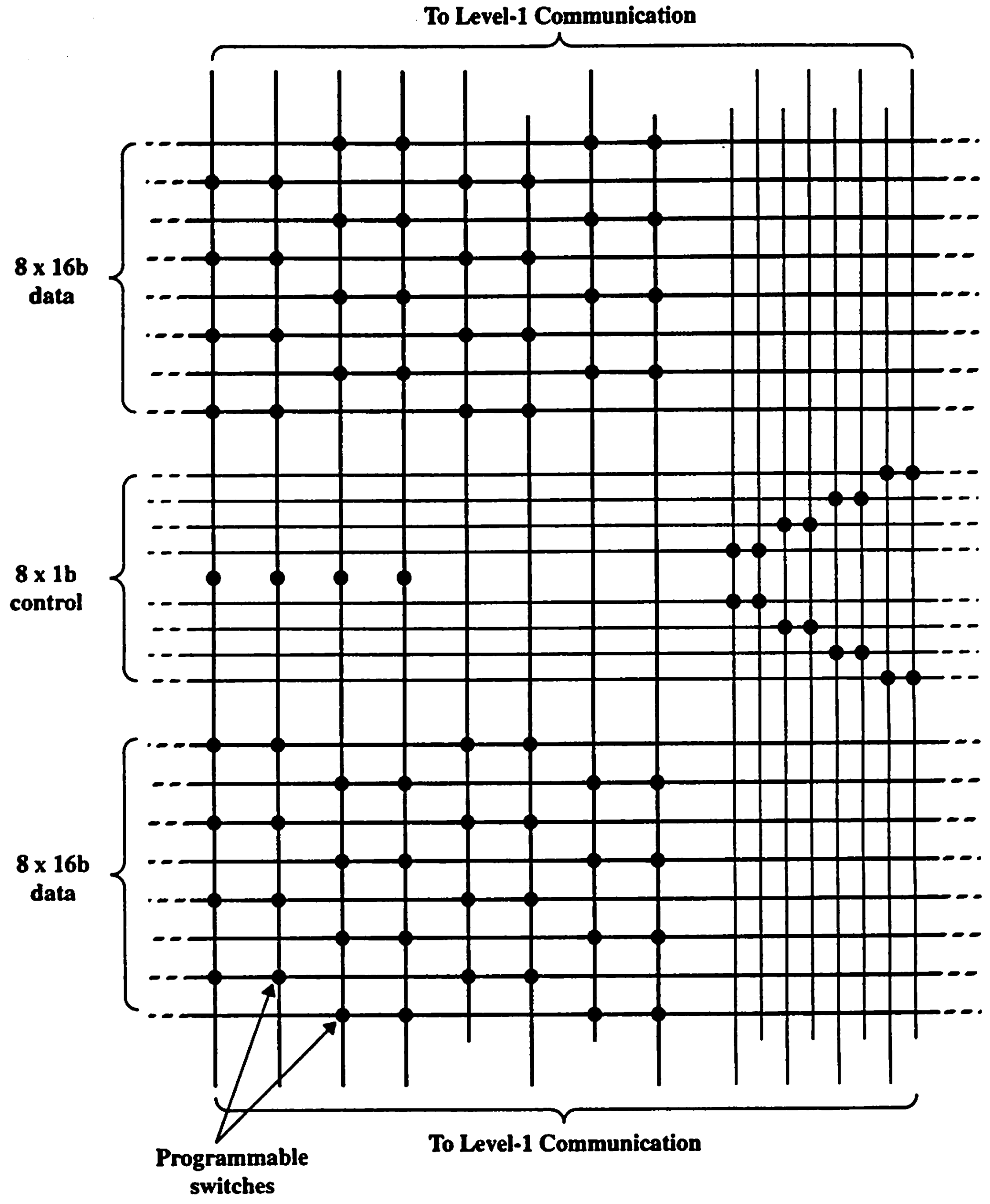


Figure 5-9: Switch Matrix between Level-1 and Level-2 Communication Network.

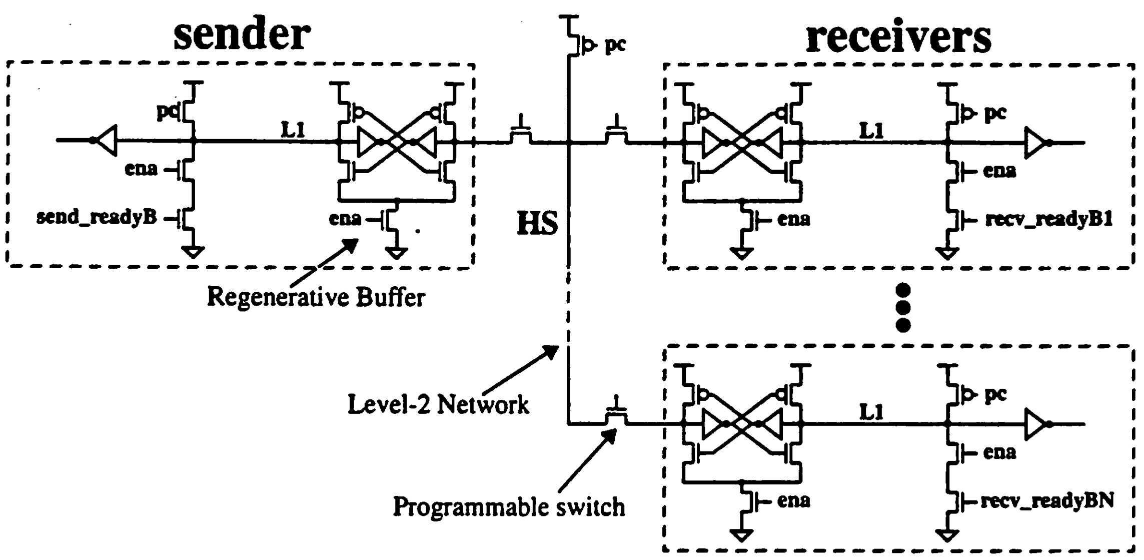


Figure 5-10: Dynamic WIRE-AND Handshake Circuit supporting

evaluation phase, i.e., the sender and all the receivers are ready. Special attention is given to the spacing of the HS lines in the layout and detailed SPICE simulations were performed to prevent capacitive coupling between neighboring HS lines from causing false discharge. A regenerative buffer is inserted between the two levels of the interconnect network to improve speed.

Switches in the network are implemented using N-channel devices only to save area and to reduce capacitive loading on the bus. This results in a long low-to-high delay, which is addressed by pre-conditioning the data buses to high before driving data onto the bus. Communications through the network take 1 clock cycle uniformly regardless of the positions of the source nanoprocessor and destination nanoprocessor(s).

Figure 5-11 shows the results of SPICE simulation of the propagation delay through the 2-Level communication network. The transferred data propagates from a Level-1 network to the Level-2 network before reaching the destination Level-1 network in another cluster. The total

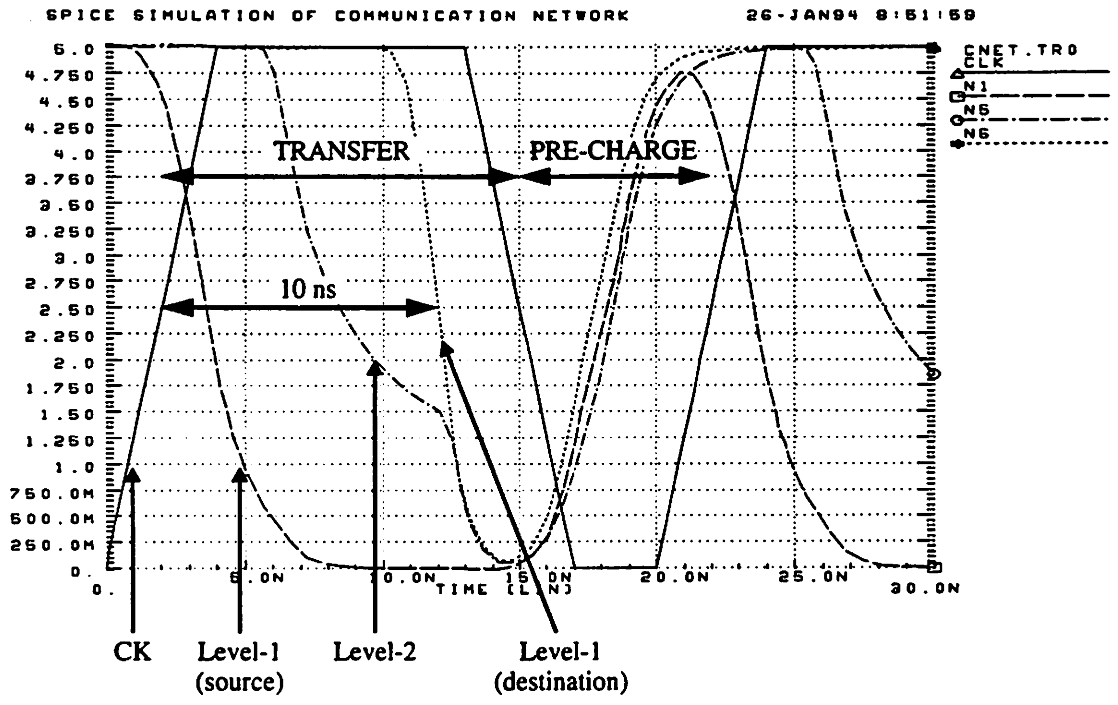


Figure 5-11: SPICE simulation of the propagation delay of the PADDI-2 communication network.

propagation delay is 10 ns which is fast enough for a transfer rate of 50 MHz using a 66% duty-cycle clock.

Using a compact layout style, the size of the Level-2 communication network is limited only by the pitch of the second-layer metal. The area efficiency is evident from the fact that the Level-2 communication network takes up only 17% of the total core area on the PADDI-2 chip. Details of the configuration of the communication network will be covered in Chapter 6 when examples of the mapping of several DSP algorithms onto the PADDI-2 chip are presented.

5.4 Scan Unit

The functionality of the PADDI-2 chip is not defined until the instruction stores of the nanoprocessors are initialized and the communication networks are configured. The scan unit is the on-chip hardware unit responsible for programming or configuring the PADDI-2 chip at system start-up. Its primary functions are summarized as follows:

- initializing the contents of the instruction memory of each nanoprocessor;
- establishing communication channels between nanoprocessors by configuring the programmable switches in the communication network;
- initializing the program counters that point to the first instructions to be executed by the nanoprocessor;
- configuring the direction of data flow (input or output) for each I/O nanoprocessors (NIOP's);
- configuring whether an I/O port is connected to another I/O port of a PADDI-2 chip or a memory;
- presetting the contents of the data buffers in each nanoprocessors;
- facilitating testing and debugging by capturing the inputs to the data buffers of each nanoprocessors, which can later be scanned out for observation.

The total length of the scan chain in the 48-nanoprocessor chip is 7464, with 143 scan flip-flops in each of the 48 nanoprocessors, and 50 scan flip-flops in each of the twelve clusters for configuring the communication patterns between the clusters and the Level-2 communication network.

Since the PADDI-2 chip is programmed only once at start-up, a 4-bit serial scan bus is used to conserve valuable pins for data I/O and reduce area overhead imposed by the routing of a parallel programmable bus. The 4-bit serial bus interface is very similar to the IEEE P1149.1 JTAG serial scan interface mechanism [54]. It consists of a bus enable signal, a mode control signal for specifying the various modes of operation and an input data line and an output data line for transferring data to and from the on-chip scan chain. Three modes of operation: SCAN, UPDATE and SINGLE-STEP, are supported.

The SCAN operation is used to preset the contents of the scan chain by serially shifting data into the scan chain using the input data line. The same operation can also be used to shift out the previous contents of the scan chain for testing and debugging.

Once the contents of the entire scan chain is set up, an UPDATE operation can be issued to transfer the contents of the scan chain to the instruction stores and data buffers. The write addresses of the instruction store and data buffers are specified in each nanoprocessor by the program counter and the write pointers which are also parts of the scan chain. All together, eight pairs of SCAN and UPDATE operations are needed to initialize the eight-word instruction store.

The SINGLE-STEP operation is implemented to assist the tedious tasks of testing and debugging the complex PADDI-2 multiprocessor chip. After the PADDI-2 chip under test is completely configured, it is put into a halt state by asserting an external HALT pin. The chip can be enabled to run for exactly one cycle by issuing a SINGLE-STEP operation using the scan bus. Inputs to each nanoprocessor in the cycle are captured in the scan chain and can be scanned out for examination using a SCAN operation. In addition to the inputs, the program counter and the full/empty status of the data buffers of each nanoprocessors are also accessible through the scan chain. Thus by issuing pairs of SINGLE-STEP and SCAN operations, snapshots of the operation of the PADDI-2 chip executing a complex program can be taken and examined easily to aid debugging.

A drawback of this scan bus design is the slow speed due to the serial nature of the bus interface. For example, to fully configure the 48-nanoprocessor chip, at least 2 ms (plus any data transfer overhead) is needed. For those applications involving several phases of computations, the chip cannot be quickly reprogrammed to handle the different tasks. Future versions of the PADDI-2 chips with improved programming bus interface are being examined.

CHAPTER 6

PADDI-2: Development System

In this chapter, the programmer model and an assembler language for the PADDI-2 architecture are first reviewed. We then describe an application development system for PADDI-2 which includes an assembler, a VHDL-based simulator for system simulation and debugging, and a VME board for hardware demonstration and testing. The architecture of a multi-PADDI-2 board currently being designed will also be mentioned. The chapter is concluded by presenting examples of the mapping of practical DSP algorithms onto the PADDI-2 architecture and some performance benchmark results.

6.1 PADDI-2 Programming Model

To the programmer, a PADDI-2 system consists of many nanoprocessors and memories interconnected by communication channels as discussed in Section 4.2.2. A nanoprocessor or a memory can communicate with other nanoprocessors and memories by explicitly sending data or control tokens via the channels. Each nanoprocessor executes an assigned program and is independent of the other nanoprocessors. The programming resources for writing a nanoprocessor

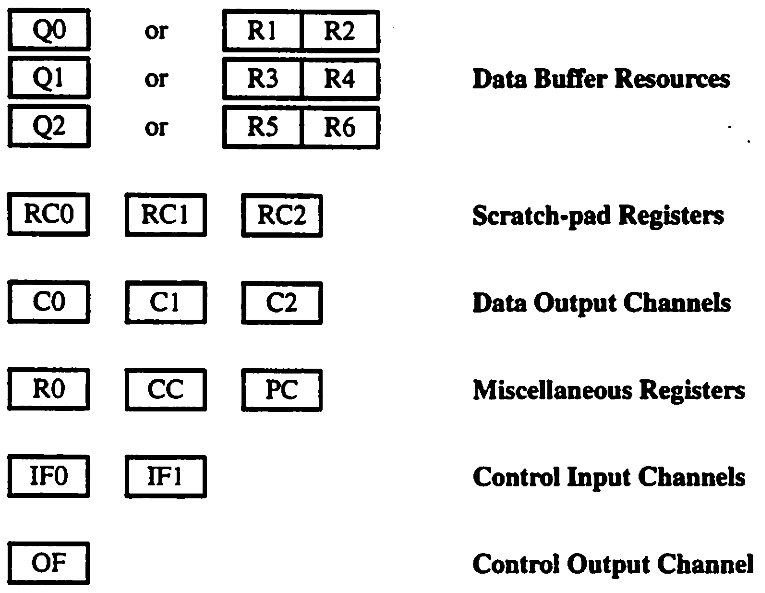


Figure 6-1: Programming Resources of Nanoprocessor.

program are discussed in this section. In the next section, the usage of these resources will be clarified when the syntax of the assembler language developed for PADDI-2 is explained.

As depicted in the nanoprocessor block diagram in Figure 5-3, each nanoprocessor is equipped with three data buffers: Q0, Q1 and Q2, two control token buffers: IF0 and IF1, three data output channels: C0, C1 and C2, and one output control token channel: OF. As shown in Figure 6-1, each data buffer qi can be configured as a queue (Q) or a small register file (Ri's). When configured as a queue (Qi's), the data buffer acts as an infinite source of data to the programmer. The datum can be read either once and then discarded or the same datum can be read many times until it is discarded. When used a register-file, a data buffer becomes a two-word random-access register-file. With three data buffers, up to six registers, R1 to R6 are available. Unlike the queues, which can only be used as source operands, these registers can be read and written to and therefore can be used as both source operands and destination operands. A dummy register, R0, which can only be used as a destination operand, is provided so that the result of the operation can be discarded when R0 is specified as the destination operand.

Besides the above register resources, three scratch-pad registers, RC0, RC1 and RC2 are provided. These scratch-pad registers can be viewed as the output ports of the ALU, i.e., the outputs of the ALU are automatically written into these registers after each instruction cycle. Since their contents are already defined by the operation of the instruction, these scratch-pad registers cannot be used as destination operands. Although only one output is needed for most operations, 3 scratch-pad registers are provided to facilitate the pipelined implementation of the multiplication operation using multiple nanoprocessors. In this case, RC0, RC1 and RC2 will contain the partial sum, the multiplicand and the multiplier of the multiplication operation, respectively.

The data output channels Ci's are used when data are communicated to another nanoprocessor or memory using the communication channels. The Ci's can be considered as the *entrances* to the communication channels. Data written into the communication channel will emerge in the same order at the receiving end of the communication channel.

In addition to the register resources and I/O resources, an internal program counter, PC and a condition code register, CC are provided. The PC points to the current instruction to be executed and can be modified by a branch instruction. CC provides a buffer for the ALU condition codes or control tokens and can be used to control a conditional branch.

The control input/output channels, IF0, IF1and of are very similar to their data counterparts, Qi's and Ci's except that the former handle control tokens for branching and the latter are used for data.

6.2 PADDI-2 Assembler Language

To ease the task of writing a nanoprocessor program, an assembler language is specially developed for the PADDI-2 architecture. In this section, the syntax of the assembler language will be introduced and the usage is illustrated using a code written for a counter. More information on the mnemonics of the assembler language can be found in Appendix A.2.

As shown in Figure 6-2, a PADDI-2 assembler program consists of two main sections: the initialization section and the program section. In the initialization section, there are one or more directives starting with the key word *.init*. The directives are used to specify the static aspect of the program. A complete listing of the directives defined is given in Table 6-1.

```
/* Beginning of Initialization Section */
.init pc=Label
[.init ...]
...
...
...
[.init ...]
 /* End of Initialization Section */

/* Beginning of Program Section */
Label1:
dest_operand = op_code(src_operand1,...,src_operandN),        /* op_code field */
[cc = c/s/x],                                                 /* cc_code field */
[c0 = rc0], [c1 = rc0/rc1], [c2 = rc2],                       /* data output */
[of = 0/1/cc],                                                /* control output */
next = Label1; /* unconditional branch */                     /* next PC field */
or
next = {cc/if0/if1}? Label1 : Label2; /* conditional branch */  /* next PC field */

Label2: . . . .

/* End of Program Section */
```

Figure 6-2: Program Structure of a PADDI-2 Assembler Program.

Table 6-1 : Listing of Directives.

Directive Name	Value	Remark	Mandatory/ Optional
PC	Label	Initial Program Counter Value	Mandatory
q0, q1, q2	r / q	Data Buffer Configuration	Mandatory
R1-R6	constant	Register initialization	Optional
oena	T / F	Output Enable (used by I/O nanoprocessors only)	Mandatory for I/O nanoprocessor
mem_ena	T / F	Memory Enable (used by I/O nanoprocessors only)	Mandatory for I/O nanoprocessor.

The PC directive specifies the very first instruction to be executed by the nanoprocessor after start-up. The q0-q2 directives set the configuration of the data buffers. The Ri's directives are used to initialize the contents of the data buffers. In the case where the program is written for an I/O nanoprocessor, the oena directive specifies the input/output direction of the I/O port and the mem_ena directive controls whether the I/O port is connected to a memory or another PADDI I/O port.

Following the initialization section is the main program section containing the nanoprocessor instructions used in the program. Referring to Figure 6-2, each instruction can be identified by a unique label and can be divided into five fields:

• op_code:	operation code field
• cc_code:	condition code field
• data output:	data token output control field
• control output:	control token output control field
• next PC:	next program counter field

```
/* Sample Assembler Code: COUNTER.s *

.init q0=r,  q1=r,  q2=r          /* All 3 data buffers are configured as
                                  register files */
.init r1 = 10                     /* loop count */
.init r3 = 1                      /* loop count decrement */
.init pc = START                  /* pc initialization */

START:      r2 = sub(r1,r3),  cc0 = s,  next = cc0? START : LOOP;
LOOP:       r2 = sub(r2,r3),  cc0 = s,  next = cc0? START : LOOP;
```

Figure 6-3: Sample PADDI-2 Assembler Code implementing a counter.

The op_code field specifies the source operands, destination operand as well as the operation to be performed. The cc_code field sets the cc register (cc) to the ALU condition code which can either be the carry output of the ALU (c), the sign bit of the ALU result (s) or the sign change signal of the ALU (x). The x condition code is provided to simplify the task of zero-detect. By subtracting one from the source operand, x is assert if the source operand is zero before the subtraction. The data and control output fields specify if data or control tokens are to be sent to other nanoprocessors via the output channels: C0, C1, C2 or OF. Lastly, the next PC field handles the control flow of the program. In the case of unconditional branch, the next PC is simply set to the instruction label specified in the next PC field. In the case of conditional branch, the next PC can be either one of the two instruction labels specified depending on the branch condition, which is either the cc register or one of the two input control queues, IF0 or IF1.

To illustrate the syntax of the PADDI-2 assembler language, the code for a down-counter is shown in Figure 6-3. In this example, all three data buffers are configured as register files and the starting PC points to the instruction labelled START. R1 is initialized to the counting period and R3 is set to the count decrement. After executing the *setup* START instruction once, the program proceeds to execute the LOOP instruction repetitively until the count becomes negative as indicated by the sign bit. At that time, the program jumps back to instruction START and restarts the counting sequence again.

6.3 PADDI-2 Application Development System

To speed up the task of developing and debugging applications on PADDI-2, a development environment comprised of a software system and a PADDI-2 hardware system was developed. In this section, the software development system based on an assembler and a VHDL simulator is described. The design of the hardware prototyping system will be presented in the next section.

The major components of the PADDI-2 software development system are shown in Figure 6-4. The entry point of the development system is a Viewlogic schematic [103] which consists of four architectural components: nanoprocessors, memories, input ports and output ports. An example of a Viewlogic schematic for the hidden surface processor application described in Section 2.2 is given in Figure 6-5. Associated with each nanoprocessor in the schematic is a pointer to a file containing the nanoprocessor assembler program to be executed by the corresponding nanoprocessor. Similarly a file pointer is also associated with each input port or output port and it specifies the file from which input data can be read or to which output data are written. The schematic is parsed by a program called PGEN (an acronym for program generator) to produce netlist information such as the number of nanoprocessors used and how they are interconnected. Using the netlist information, an assembler compiles the user defined nanoprocessor programs into PADDI-2 object code which is an intermediate code format used by the VHDL nanoprocessor model. Besides user defined programs, users can also make use of pre-compiled programs such as counter, adder and multiplier stored in a library. Next a VHDL model describing the system is created using the netlist information, nanoprocessor object codes and VHDL models of the components. The system VHDL model can then be simulated using a VHDL simulator, e.g., the Synopsis VHDL simulator [98], to verify the functionality.

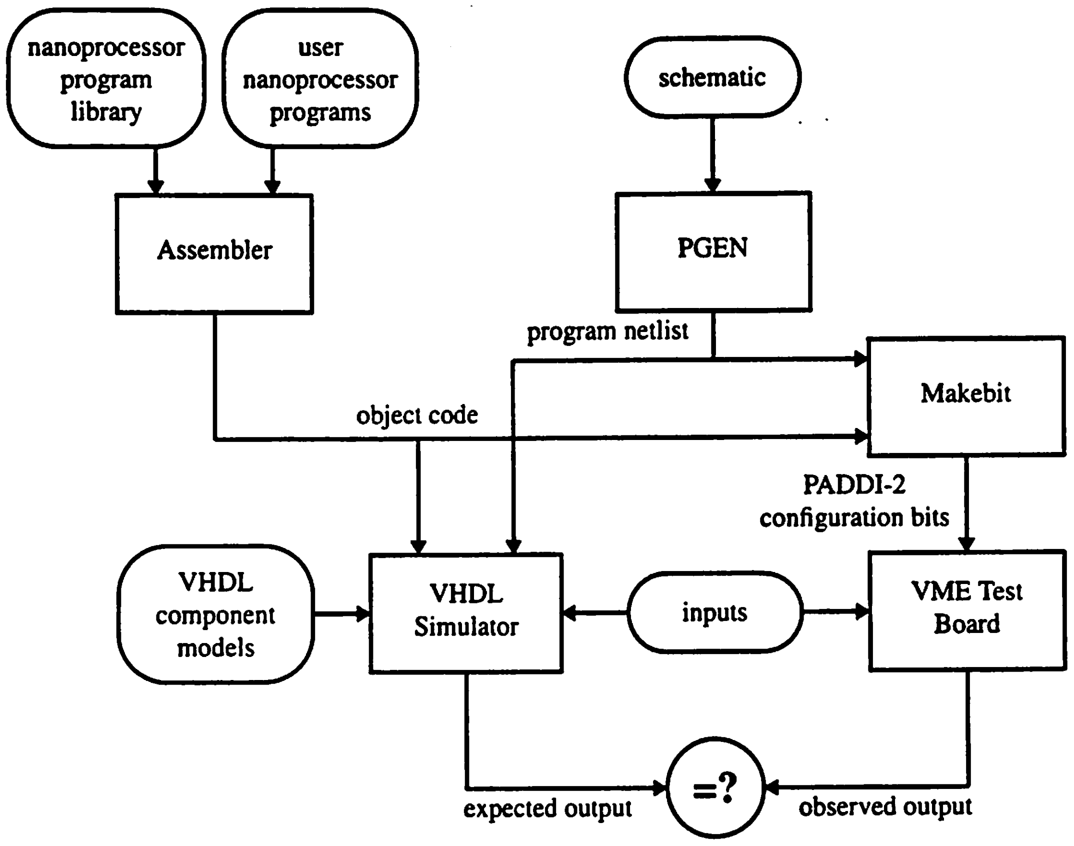


Figure 6-4: PADDI-2 Software Development System.

After the system is debugged and verified, the *makebit* program can be called upon to generate the configuration bit pattern for programming up a PADDI-2 chip. The configuration information is then downloaded onto a PADDI-2 chip on a VME board to implement the system in hardware.

As an added bonus, the same software system can also be used to develop test programs for verifying the functionality of PADDI-2 chip. A test program is first simulated using the VHDL simulator to generate the *expected output* from the chip. The same program can then be downloaded to the PADDI-2 chip under test and the outputs produced by the hardware can be

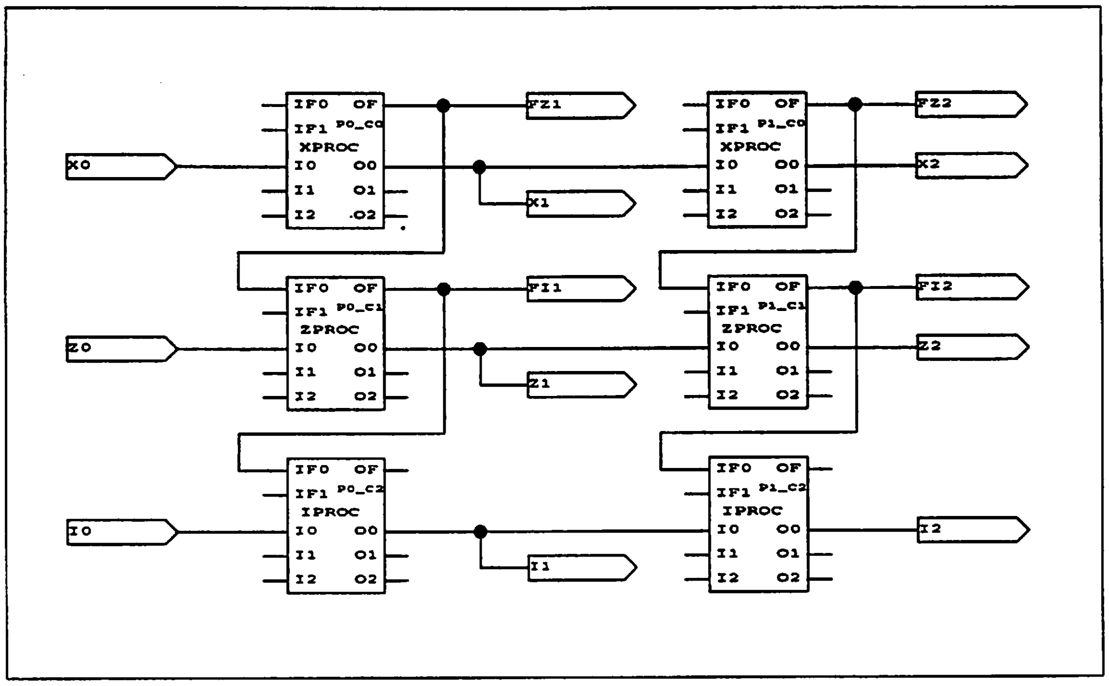


Figure 6-5: Viewlogic Schematic of the Hidden Surface Processor

easily observed by making use of the SCAN and SINGLE-STEP capabilities described in Section 5.4. The *observed output* can then be checked against the expected output to verity the functionality of the test chip.

6.4 PADDI-2 DSP Prototyping Hardware System

To demonstrate the PADDI-2 architecture and to test the PADDI-2 prototype chip, a simple hardware system as shown in Figure 6-6 is developed. The system consists of a host workstation, a Heurikon single-board CPU [48] and several PADDI-2 board housed in a VME-cardcage. The host is a SUN SPARC workstation and functions as a software development station and as a large auxiliary data storage. It is connected to the CPU board in the VME-cardcage via an ethernet. The CPU board, based on a MC68000 microprocessor running the VxWork light-weight real-time

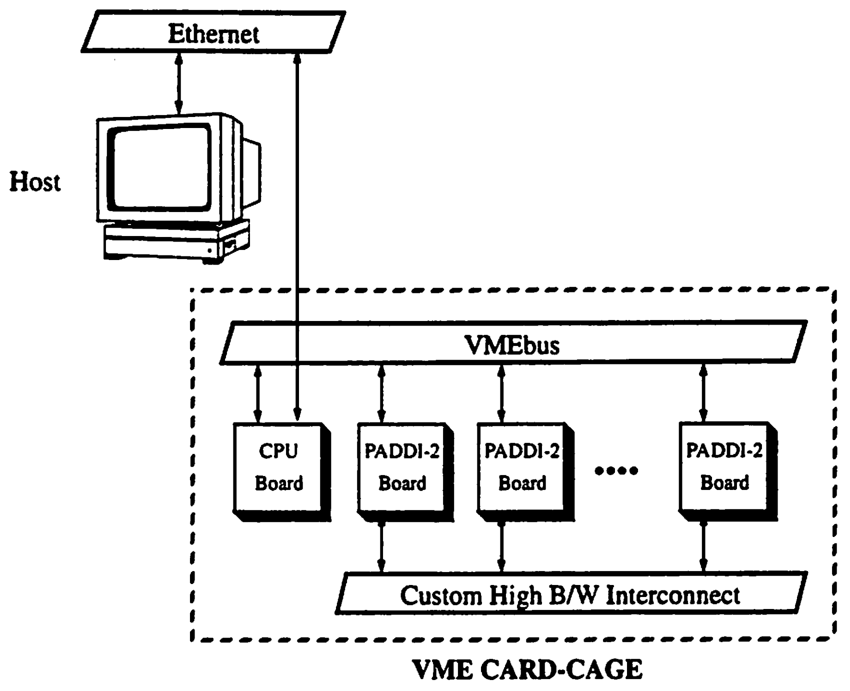


Figure 6-6: PADDI2 DSP Prototyping Hardware System.

kernel [108], serves as an interface between the host workstation and the PADDI-2 VME board. The CPU board and the PADDI-2 boards are connected by a 32-bit VMEbus [104].

As shown in the block diagram in Figure 6-7, the VME board includes a VME interface block, a board controller, a PADDI-2 chip and a 64K x 16-bit memory module. The VME interface block consists of an address decoder and a VME2000 slave controller chip [76]. The address decoder compares the address broadcasted on the VMEbus with an ID unique for each board. If matched, the VME2000 chip which monitors and generates the necessary control interface signals with the VMEbus will be enabled to handle the bus transaction. A Read (Write) bus transaction is completed when the data is read from (written to) an internal register of the board controller.

To simplify design changes and debugging of the board, an Xilinx 3090 field-programmable

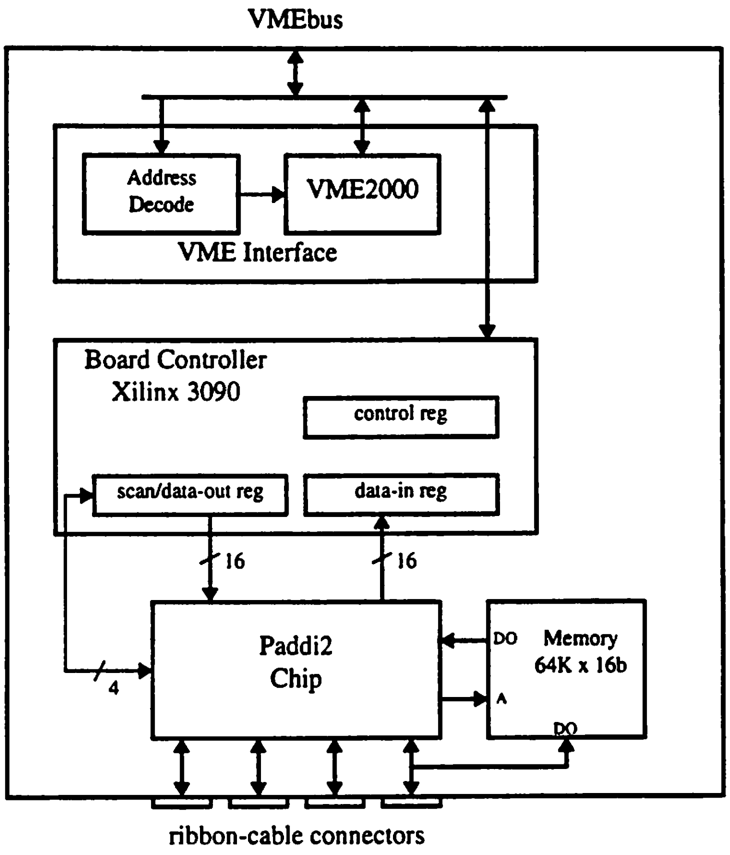


Figure 6-7: PADDI2 VME Board.

gate array chip [110] is used to implement the board controller. On one side, the board controller is hooked up to the VMEbus through which the internal registers of the board controller can be programmed. On the other side, the board controller is connected to the PADDI-2 chip by a 4-bit scan bus, a 16-bit input data bus and a 16-bit output data bus.

Three programmable registers: the control register, the scan/data-out register and the data-in register, are provided in the board controller. By setting or resetting bits in the control register, different operations, e.g., Scan-Enable and Write-Scan-Register, can be triggered from the host. The scan/data-out register is used for storing data to be scanned into or out of the PADDI-2 chip scan chain. Lastly, the data-in register is provided for temporary buffering of data received from the PADDI-2 chip. More detailed descriptions of these registers are given in Appendix A.3.

The PADDI-2 chip can be programmed by the board controller using the 4-bit scan bus. Two of the eight I/O buses of the chip are connected to the board controller so that data can be directly sent from or received by the host. Another three I/O buses are connected to the data input port, data output port and address port of the memory module. To simplify the I/O bus design, the current PADDI-2 chip can only interface with memories with separate data I/O. With an enhanced design of the I/O bus, future versions of the PADDI chip can support memories with shared I/O to reduce I/O bandwidth requirement. The remaining three I/O buses, as well as the I/O bus connected to the data input port of the memory module, are routed to four ribbon-cable connectors to allow communication between PADDI-2 chips on different boards.

Including a 2.5 x 6 inch^2 prototyping area, the board measures 10 x 11 inch^2 and is implemented using a 4-layer printed circuit board. A layout of the board can be found in Appendix A.4.

6.5 Multi-PADDI-2 Board

The main functions of the PADDI-2 VME board described above are chip testing and demonstration of simple DSP algorithms. Many boards are needed to provide enough processing power for more complex algorithms. To improve the computation throughput and expand the application scope of the system, a printed circuit board consisting of eight PADDI-2 chips is being designed. The design of the board will be briefly described in this section.

As shown in the block diagram in Figure 6-8, the multi-PADDI-2 board contains an Sbus interface [97], an video I/O block and a PADDI-2 array interconnected by a 16-bit local bus. The L64853 Sbus controller chip from LSI Logic, Inc. [66] handles communication between the host SUN workstation and the board using an Sbus. It is also responsible for the programming of the

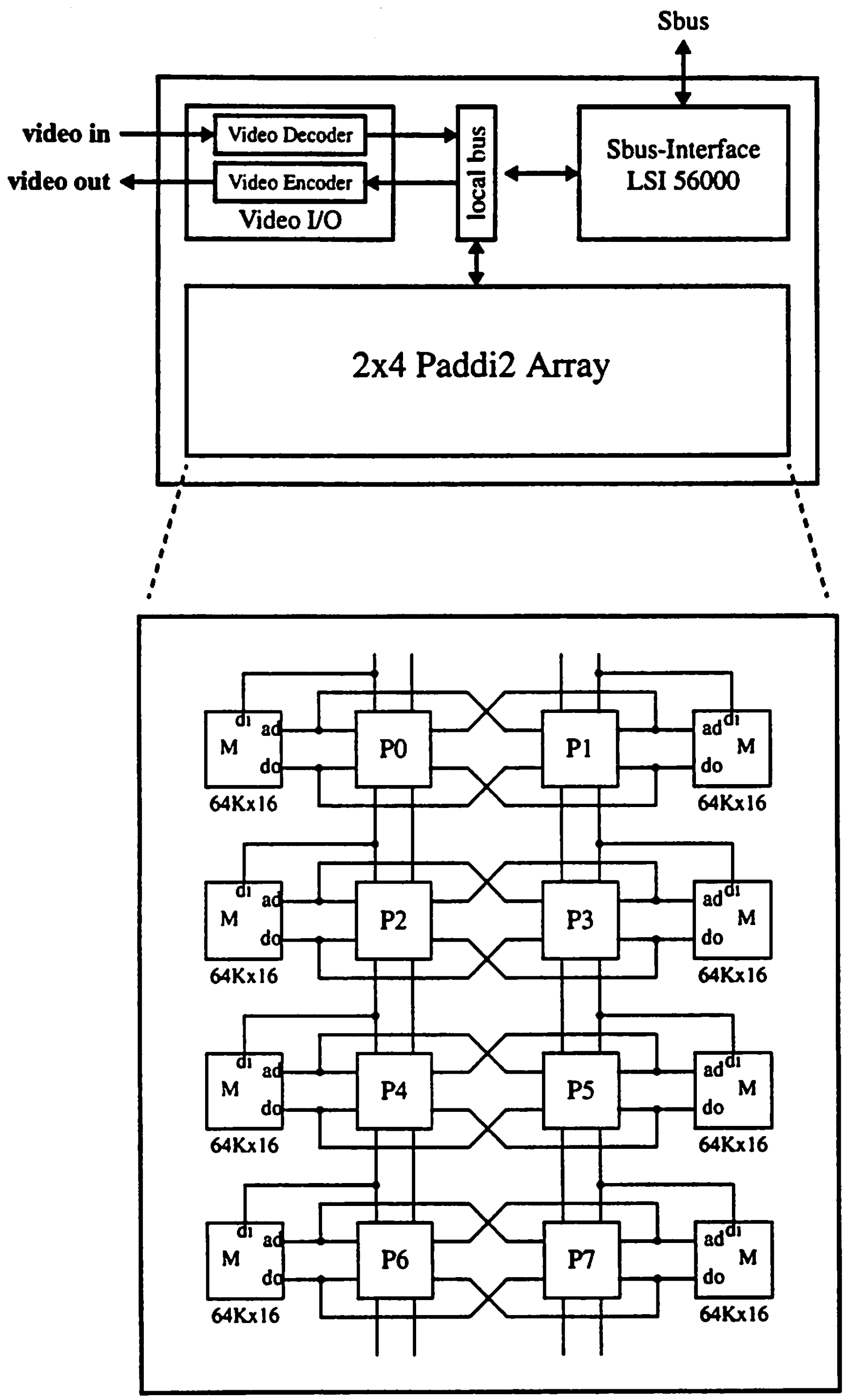


Figure 6-8: Block Diagram of the Multi-Paddi2 Board.

PADDI-2 chip array. The video I/O block, consisting of a video encoder and a video decoder, provides real-time video I/O capability for the system. Data can be easily transferred between the controller chip, the video I/O block and the PADDI-2 array using the local bus.

Computation power is provided by the PADDI-2 array consisting of eight PADDI-2 chips and eight 128-Kilobyte memories. Using a 2x4 torus interconnection pattern, each PADDI-2 chip can communicate with each of its four immediate neighbors using two I/O buses. A PADDI-2 chip can also be configured to interface with a memory chip to cope with memory intensive applications. Four buses at the edge of the array are hooked up to four connectors for connection to other boards. Running at 50 MHz, each board can provide up to 20 Giga operations per seconds peak. The design of the board is still in progress at the writing of this thesis. Further details of the board architecture will be provided in future publications.

6.6 Mapping of DSP Algorithms onto PADDI-2

To illustrate the use of the PADDI-2 development system presented in the previous sections of this chapter, detailed examples of the mapping of two practical DSP algorithms onto the PADDI-2 architecture will be discussed in this section. The two algorithms used are the Hidden Surface Processor and the Viterbi Detector, both of which have been explained and analyzed in Chapter 2 as target DSP algorithms of PADDI-2.

6.6.1 Hidden Surface Processor

The Hidden Surface Processor uses the Z-buffer algorithm for elimination of hidden surfaces in 3-D graphics on a raster scan-line basis. The operations involved in the algorithm are shown in Figure 6-9 and further details of the algorithm can be found in Section 2.2.

The logical mapping of the algorithm onto an linear array of nanoprocessors is described in Figure 6-10. Each pixel processor is implemented by three nanoprocessors, namely Xproc, Zproc

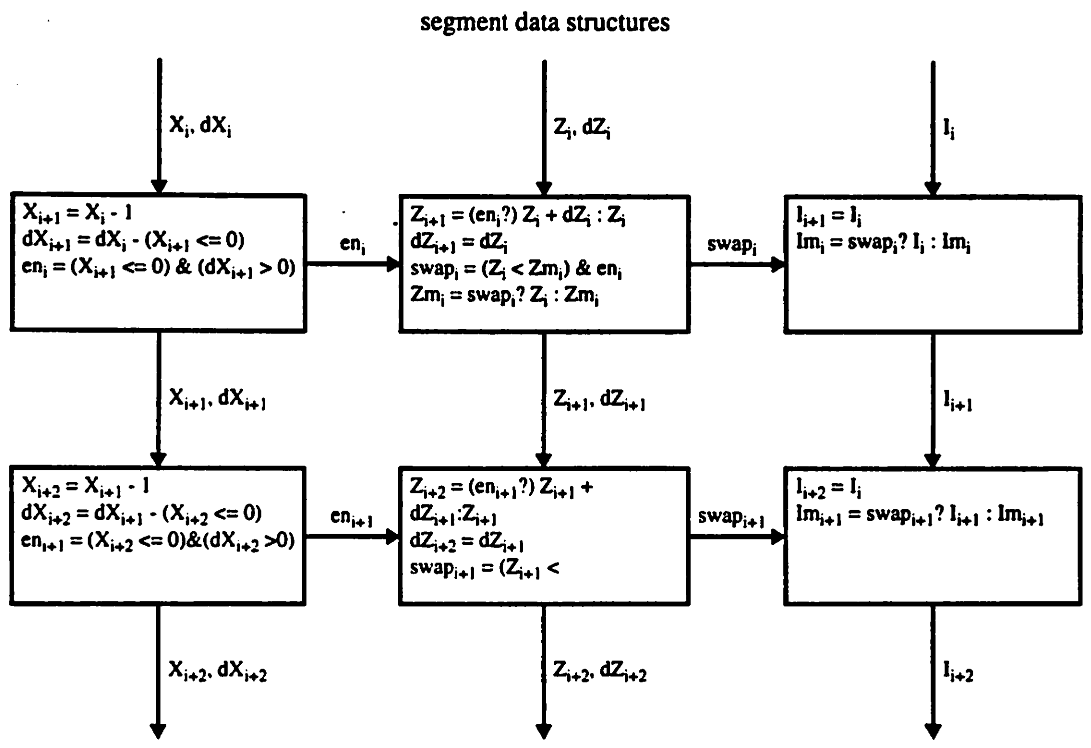

segment data structures
X_i, dX_i
Z_i, dZ_i
I_i
$X_{i+1} = X_i - 1$
$dX_{i+1} = dX_i - (X_{i+1} <= 0)$
$en_i = (X_{i+1} <= 0) \& (dX_{i+1} > 0)$
en_i
$Z_{i+1} = (en_i?) Z_i + dZ_i : Z_i$
$dZ_{i+1} = dZ_i$
$swap_i = (Z_i < Zm_i) \& en_i$
$Zm_i = swap_i? Z_i : Zm_i$
$swap_i$
$I_{i+1} = I_i$
$Im_i = swap_i? I_i : Im_i$
X_{i+1}, dX_{i+1}
Z_{i+1}, dZ_{i+1}
I_{i+1}
$X_{i+2} = X_{i+1} - 1$
$dX_{i+2} = dX_{i+1} - (X_{i+2} <= 0)$
$en_{i+1} = (X_{i+2} <= 0)\&(dX_{i+2} >0)$
en_{i+1}
$Z_{i+2} = (en_{i+1}?) Z_{i+1} +$
$dZ_{i+1}:Z_{i+1}$
$dZ_{i+2} = dZ_{i+1}$
$swap_{i+1} = (Z_{i+1} <$
$swap_{i+1}$
$I_{i+2} = I_i$
$Im_{i+1} = swap_{i+1}? I_{i+1} : Im_{i+1}$
X_{i+2}, dX_{i+2}
Z_{i+2}, dZ_{i+2}
I_{i+2}

Figure 6-9: Z-buffer algorithm for Hidden Surface Elimination.

and Iproc, and more pixel processors can be easily added to the system to handle longer scan-lines. While data structures of the line segments transverse the array from one pixel to another, signal flags are passed between the nanoprocessors in each pixel processor. For example, the Iproc only updates the intensity of the pixel when the Zproc sends a *swap* flag to signal that the incoming line segment is in front of the stored line segment in this pixel position. Detailed listings of the three nanoprocessor assembler programs can be found in Appendix A.5. It can be derived from the nanoprocessor programs that in the worst case, 4 cycles are required to handle one line-segment data structure and therefore the system can process 12.5 million line segments per second using a 50 MHz clock.

After the logical mapping, the system is implemented by mapping each logical nanoprocessor

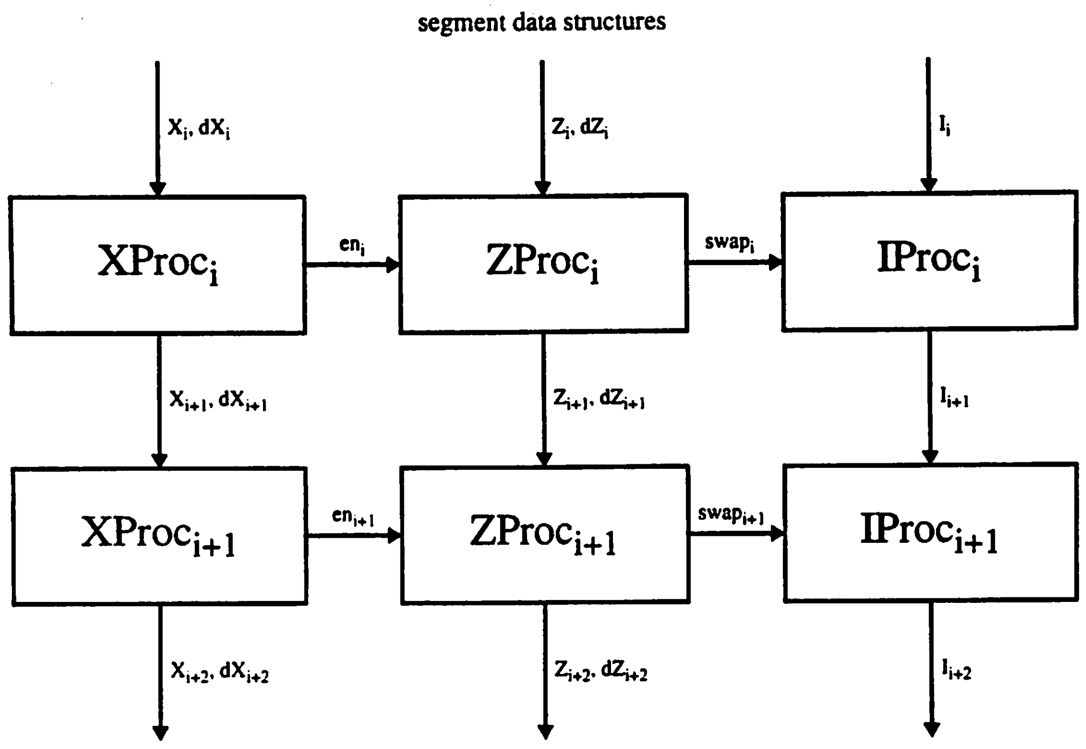


Figure 6-10: Logical Mapping of Hidden Surface Processor onto PADDI-

to a physical nanoprocessor in a PADDI-2 chip. An example of the physical mapping of a 4-pixel Hidden Surface Processor onto12 nanoprocessors is shown in Figure 6-11. Because of the systolic data flow pattern of the algorithm, the demand on communication network bandwidth is low. Most of the communications between pixel processors can be handled by bypassing the data between neighboring nanoprocessors without using the Level-1 communication network.

6.6.2 Viterbi Detector

To simplify the example, a 4-state Viterbi Detector is used as shown in Figure 6-12 but the example can be easily extended to more states by simple change in the communication network. Unlike the Hidden Surface Processor application, the data flow pattern of the Viterbi algorithm is

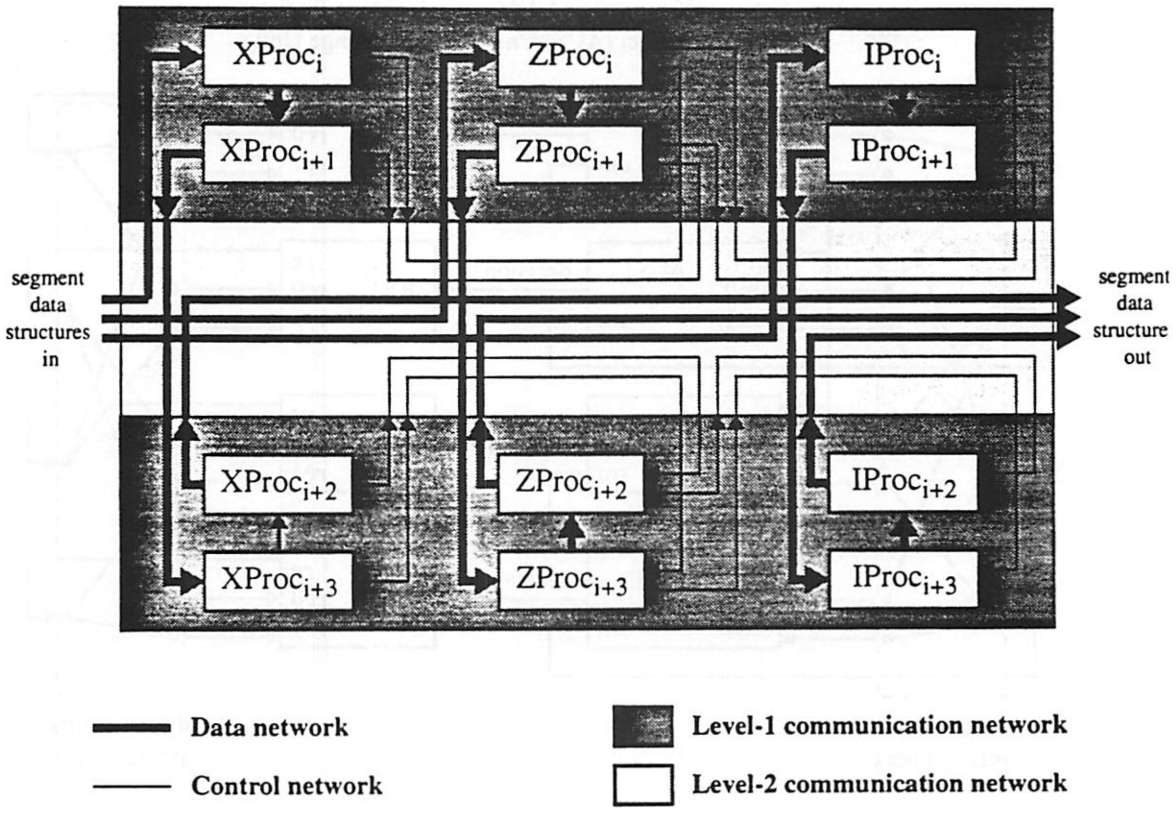


Figure 6-11: Physical Mapping of Hidden Surface Processor onto PADDI-2 Chip

very irregular because of the trellis interconnection network between the ACS units and the register-exchange units.

The logical mapping of the algorithm onto a network of nanoprocessors is shown in Figure 6-13. The design is very modular; 3 nanoprocessors are used for each state: two for the ACS operation (PE0 and PE1) and one for the register-exchange operation (PE2), and the trellis connection is handled by the communication network. All together, 12 nanoprocessors are needed to implement the 4-state Viterbi detector. A complete listing of the three nanoprocessor assembler

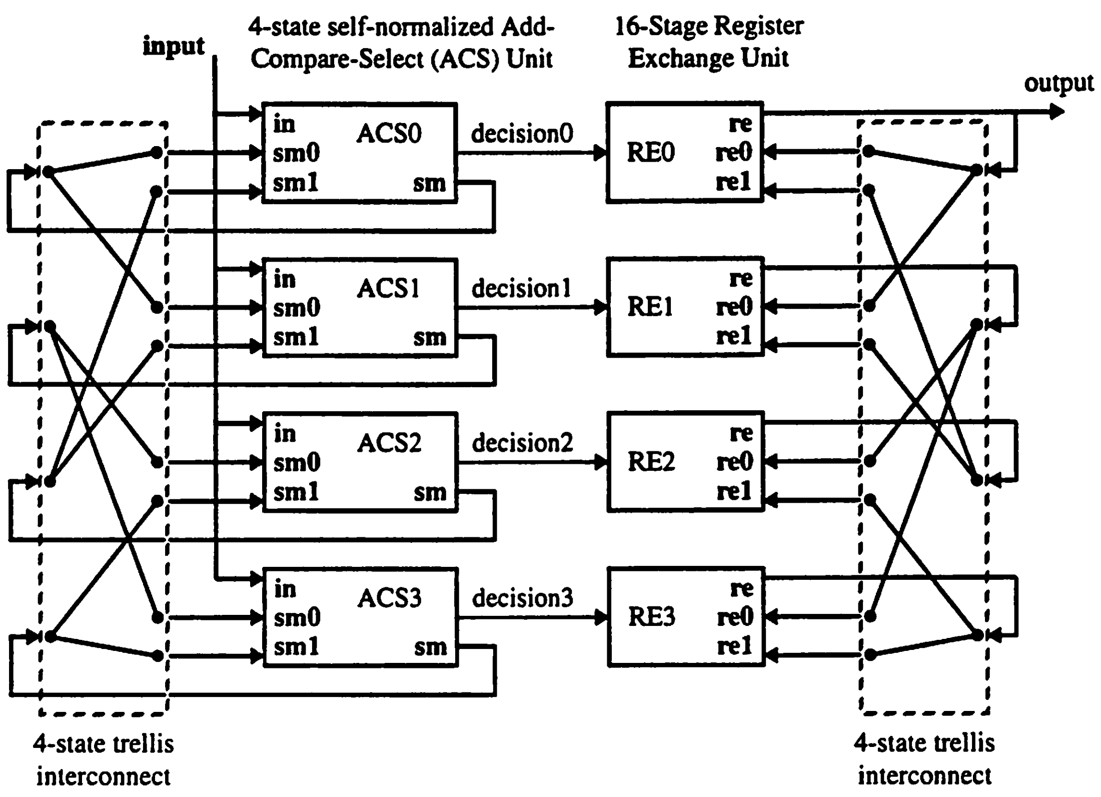

input
4-state self-normalized Add-Compare-Select (ACS) Unit
16-Stage Register Exchange Unit
output
in
ACS0
sm0
sm1
sm
decision0
RE0
re
re0
re1
ACS1
decision1
RE1
ACS2
decision2
RE2
ACS3
decision3
RE3
4-state trellis interconnect
4-state trellis interconnect

Figure 6-12: Block Diagram of 4-state Viterbi Detector.

programs can be found in Appendix A.6. Since it takes 4 cycles to implement one iteration of the ACS operation and the register-exchange operation, a decode rate of 12.5 Mbit is achieved using a PADDI-2 chip running at 50 MHz.

The physical mapping of the Viterbi detector onto the PADDI-2 chip is shown in Figure 6-14. In this 4-state example, the trellis connection is easily implemented by the Level-1 communication network. For another Viterbi detector involving more states, a combination of Level-1 and Level-2 communication networks is needed.

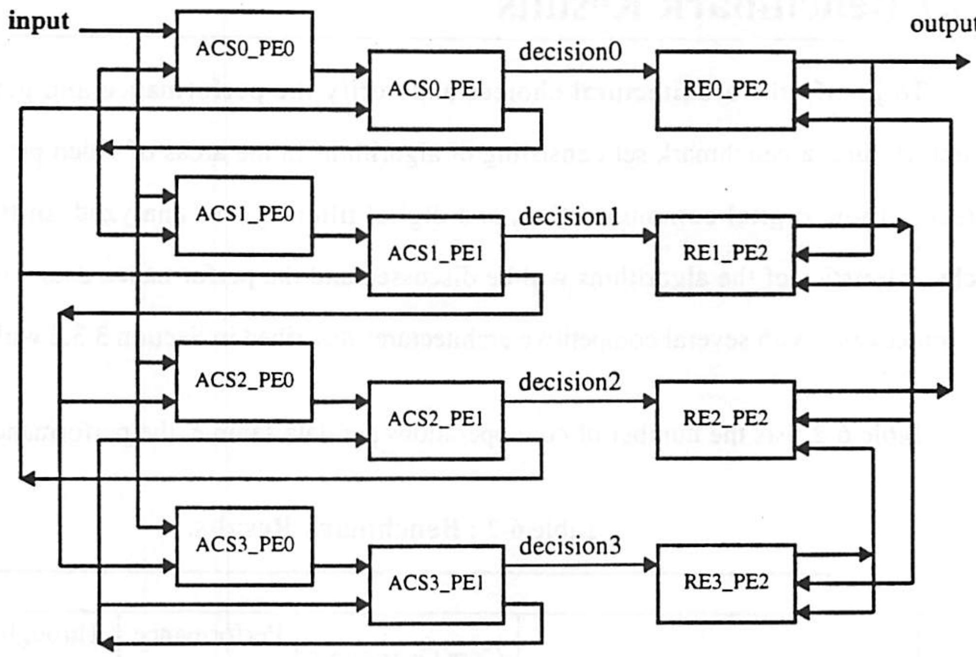


Figure 6-13: Logical Mapping of 4-state Viterbi Detector onto PADDI-2.

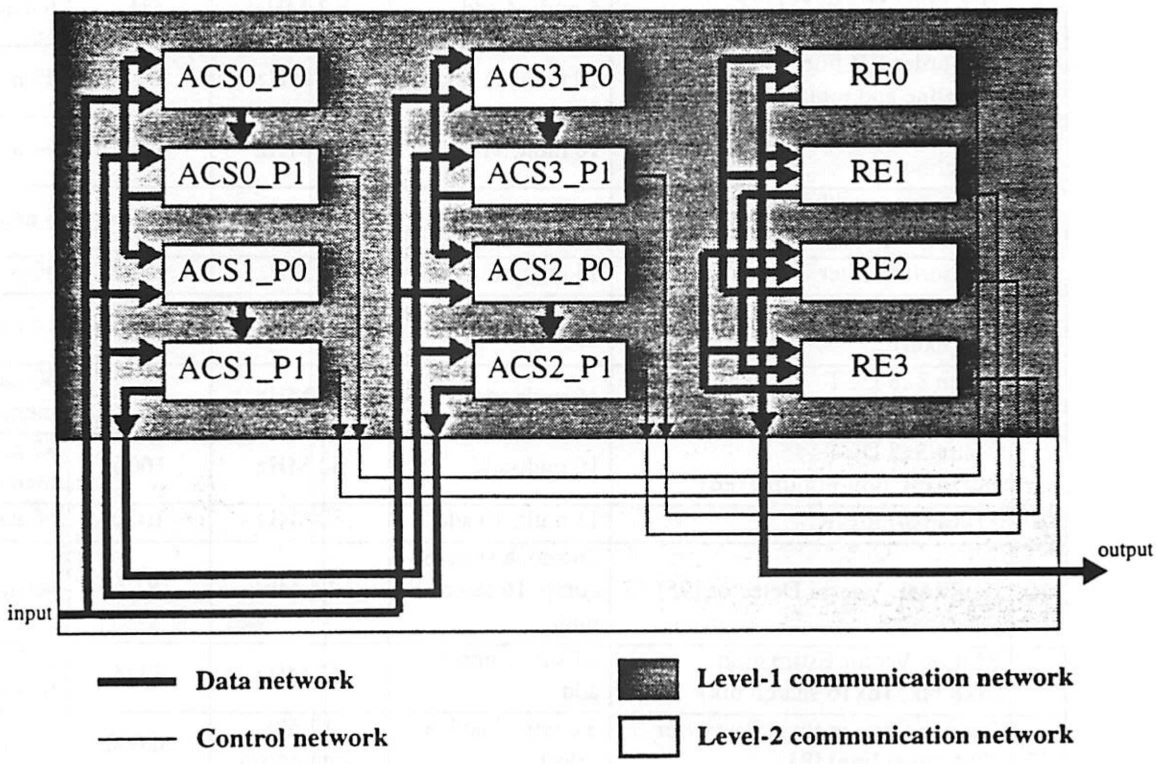


Figure 6-14: Physical Mapping of 4-state Viterbi Detector onto PADDI-2 Chip

6.7 Benchmark Results

To justify the architectural choices and verify the performance and generality of the architecture, a benchmark set consisting of algorithms in the areas of video processing, speech recognition, digital communication, and digital filtering was analyzed. In this section, the characteristics of the algorithms will be discussed and the performance data will be presented. Comparisons with several competitive architectures described in Section 3.3.2 will be made.

Table 6-2 lists the number of core operations per data sample, the performance achieved and

Table 6-2 : Benchmark Results.

#	Algorithms	Core Operations (per sample)	Performance (Sampling-Rate)	Through-put (MOPS)	Hardware
1	2nd-order IIR filter [75] (original)	5 mult, 4 add	8.3 MHz	75	6 nano
2	2nd-order IIR filter [75] (pipeline and multiplexed)	10 mult, 11 add	25 MHz	525	27 nano
3	2nd-order IIR filter [75] (pipeline)	10 mult, 11 add	50 MHz	1050	54 nano
4	2-dim space address generator [101]	2 add, 1 comp[a]	50 MHz	150	3 nano
5	3x3 sorting filter	15 comp & select	50 MHz	1500	30 nano
6	Video Matrix conversion [90] (8b pixel)	4 mult, 6 add	50 MHz	500	14 nano
7	2-dim 8x8 DCT (8b pixel, multiplexed)	16 mult-add	25 MHz	800	50 nano, 2 memories
8	2-dim 8x8 DCT (8b pixel, non-multiplexed)	16 mult-add	50 MHz	1600	82 nano, 2 memories
9	11-Tap 8-bit FIR	11 mult, 10 add	50 MHz	1050	56 nano
10	(E)PRML Viterbi Detector [95]	26 add, 8 shift, 8 comp, 16 select, 7 mult	12.5 Mb/s	812	40 nano
11	Motion Vector Estimation (8x8 blk, 16x16 search blk)	64 sub, comp & add	22 MHz	4224	254 nano, 4 memories
12	Hidden Surface Pixel Processor (256-pixel line) [81]	3 comp, 3 add, 4 select	12.5M segments/s	32000	640 nano

a. Compare operation.

the amount of on-chip hardware used for the benchmark algorithms. In an attempt to reflect the complexity of the algorithms, only operations involved in actual algorithmic computation are counted as *core operations*. Overhead operations and bookkeeping operations, which vary significantly depending on the implementations are excluded. All operations are 16 bit unless specified otherwise. Since most general purpose DSP processors are equipped with parallel multipliers, a multiplication is counted as only one operation in the throughput calculation for fair comparison although eight nanoprocessor operations are actually executed. The throughput requirements are very high in general and most algorithms require irregular communication patterns.

The first three algorithms in the table are three different implementations of a second-order infinite impulse response (IIR) filter and they demonstrate how algorithms with recursive bottlenecks can sometimes be transformed to allow more parallelism (more pipeline stages). The results show good correlation between performance gain and hardware cost.

Figure 6-15 shows the implementation of a 2-dimensional 8x8 Discrete Cosine Transform (DCT) at a 25 MHz sampling rate using 50 nanoprocessors and two memory banks as transpose buffers. The algorithm is decomposed into two one-dimensional DCT's and a transpose operation. The 8-bit by 16-bit multiplication is implemented in a pipeline fashion using two nanoprocessors. The transform coefficients stored in the data buffers of the nanoprocessors are streamed into the multiplication nanoprocessors together with the input pixels. The address generation unit (AG2), which is very similar to the 2-dimensional space address generator described in Section 4.2.3, produces the desired address sequence for the transpose operation.

As a comparison, a general purpose DSP processor such as the TMS320C25 can only achieve a sampling rate of 250 KHz for the 2-dimensional 8x8 DCT with optimized code [74]. By further pipelining of the multiplication operations, a sampling rate of 50 MHz can be achieved by using 82 nanoprocessors.

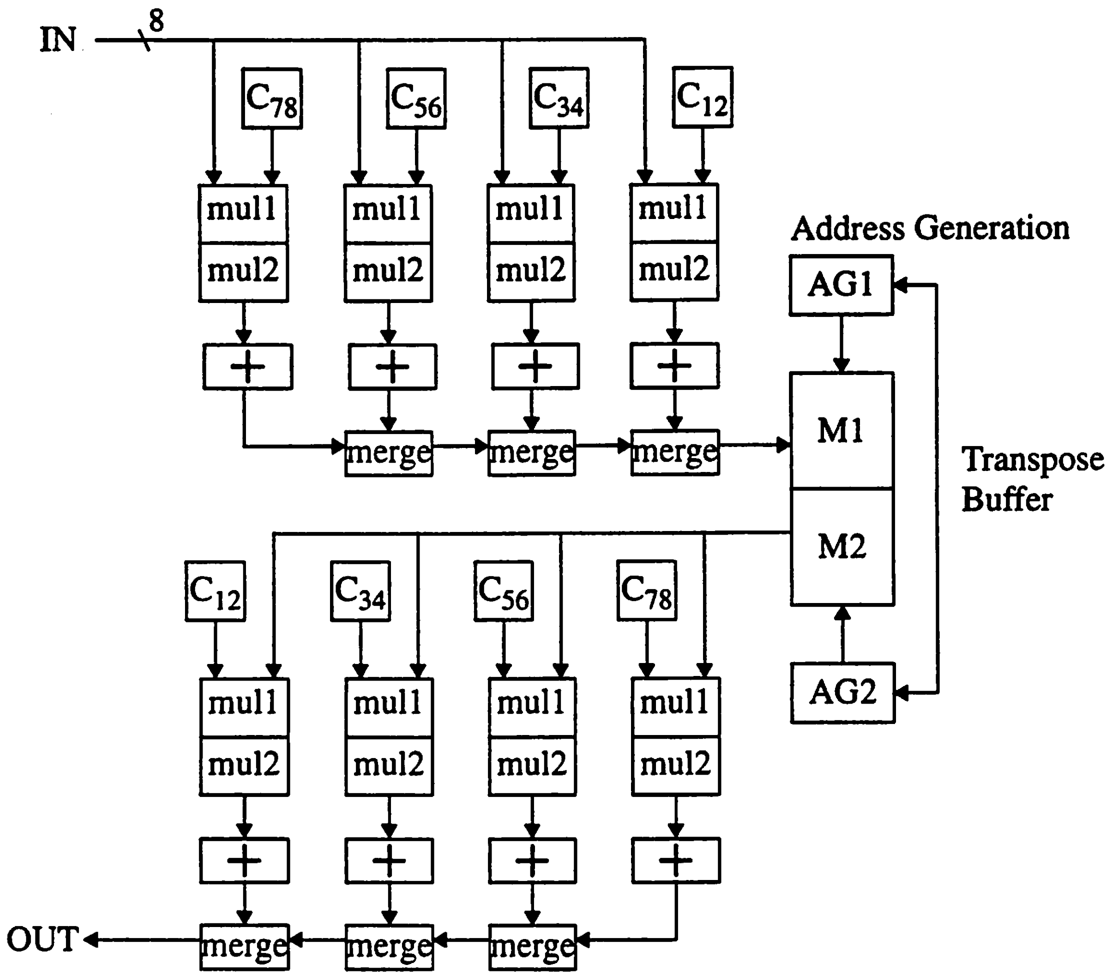


Figure 6-15: 2-D 8x8 Discrete Cosine Transform at 25 MHz sampling rate using 50 nanoprocessors.

The Viterbi benchmark implements the EPRML Viterbi detector described in Section 2.1 which includes a 7-tap transversal filter and an 8-state Viterbi detector. Due to the tight recursive loop in the ACS unit, the decode rate of 12.5 Mb/s is only a quarter of the maximum clock frequency (50 MHz). Performance can be improved by applying lookahead techniques [115] to expose more parallelism in the ACS algorithm at the expense of hardware complexity.

Other benchmark algorithms include the Hidden Surface Processor described in the previous section and the Motion Vector Estimator used in many video compression algorithms. Considering that only core operations are counted, and that the performance of a typical commercial DSP processor implemented in a technology similar to the one used by PADDI-2, is of the order of several hundred's of MOPS peak, the PADDI-2 architecture promises to be a high performance architecture for high throughput DSP algorithms.

6.7.1 Comparisons

To further gauge the performance of the PADDI-2 architecture, several DSP algorithms in the benchmark set are also implemented on two competitive architectures: the NEC Video Signal Processor and the Philips Video Signal Multiprocessor. These two architectures are chosen for comparisons because their approaches are very different from the PADDI-2's even though all three architectures target high throughput DSP algorithms. The NEC VSP achieves high performance by pushing circuits and process technologies as evident by the fast clock rate. The Philips VSP, like PADDI-2, provides high throughput using multi-processing. However, the granularity of the processing elements and programming paradigm are very different from those of PADDI-2's. More details of the two architectures can be found in Section 3.3.2. Some pertinent characteristics of the three integrated circuits are listed in Table 6-3 for reference.

Four DSP algorithms in the benchmark set are mapped to the three architectures for comparisons. The amount of hardware in terms of the number of IC chips required to achieve a certain performance is used as the figure of merits. Many subtle problems can arise when comparing very different architectures such as the three under considerations. First, it is very difficult to ensure that the implementations of the algorithms on the various architectures are all optimized to the same degree. In some cases, a re-formulation of the algorithm can produce an improved mapping onto an architecture. Moreover, some important details about the architecture may not be sufficiently covered in the literature. Consequently, we may not be aware of useful features that can boost performance, or architectural bottlenecks that can degrade performance.

	PADDI-2	NEC VSP [45]	Philips VSP [102]
Process Technology	1-μm CMOS (2-metal)	0.8-μm BiCMOS (3-metal)	0.8-μm CMOS (2-metal)
Die Size	144 mm^2	202 mm^2	156 mm^2
Data Path Width	16-bit	16-bit	12-bit
# Transistors	600 K	1130 K	1150 K
# Pins	208	132	208
Clock Speed	50 MHz	250 MHz	50 MHz[a]

Table 6-3 : Important IC parameters.

a. Since the Philips VSP uses a faster process compared to the PADDI-2 process, its clock speed is adjusted slightly from the published rate of 54 MHz to 50 MHz (the PADDI-2 clock rate) to simplify performance comparison [102].

To ensure that the performance of the two VSP architectures are not under-estimated, we made many favorable assumptions in our evaluation. It is assumed that the *peak* processing power of the two chips are always available, i.e., processing stalls due to control hazards, data dependencies, hardware resource conflicts, etc., are ignored. In cases where more than one chip are needed to deliver the performance, all the overhead associated with multiprocessing, such as communication latency and I/O conflicts, are also neglected. In other words, we hope to make sure that the results obtained for the two VSP architectures are based on the best-case scenario which may actually be unrealistic in certain cases.

Let's consider the 2-dimensional DCT algorithm discussed in the previous section as an example. Sixteen multiply-accumulate operations are processed for each pixel at the 25 MHz rate. When the algorithm is mapped onto the Philips VSP which executes four operations for each multiply operation, a total of eighty operations are required for each pixel. Given that each of the twelve arithmetic logic elements (ALE's) on-chip can execute one operation at the 50 MHz rate, a total of 40 ALE's or 3.3 chips are needed to achieve the pixel-rate of 25 MHz. This figure is the lower-bound on hardware requirement because many miscellaneous operations such as the handling of filter coefficients and transpose buffering are not considered. Moreover, extra hardware resources which may be needed in order to make the 3.3 chips work in tandem are also neglected.

The hardware requirements and the normalized results of the three architectures are shown in Table 6-3. While the PADDI-2 results were obtained by careful simulations, the results for the other architectures were either derived from published results or estimated using the favorable assumptions described above. In all cases, the PADDI-2 architecture out-performs the other two by a factor of about 2 or 3. It is interesting to note that despite the facts that the NEC VSP contains a high speed parallel multiplier and the PADDI-2 chip has none, the PADDI-2 chip is still more efficient than the NEC VSP in implementing the first two algorithms which involve many multiply operations.

DSP Algorithms	**PADDI-2** # IC's / normalized	**NEC VSP** # IC's / normalized	**Philips VSP** # IC's / normalized
2-dim 8x8 DCT, 8-b pixel (25 MHz)	1 + 2 mem[a] / 1	6.6[b] / 3.3	3.3 / 1.7
11-Tap 8-b FIR (50 MHz)	1.2 / 1	2.2 / 1.8	4.5 / 3.8
(E)PRML Viterbi Detector (12.5 Mb/s)	0.8 / 1	2.4 / 3.0	1.5 / 1.9
256-pixel HSPP (12.5 M segments/s)	13.3 / 1	128 / 9.6[c]	42 / 3.2

Table 6-4 : Comparison of Hardware Requirements.

a. Two small external memories are counted as one IC.
b. Published result [45].
c. The hardware requirement is computed assuming that the exact algorithm as described in Section 6.6.1 is used. The result can possibly be improved by reformulating the algorithm.

CHAPTER 7

Conclusions

This dissertation has presented and described PADDI-2, adata-driven multiprocessor architecture called PADDI-2 for rapid prototyping of complex DSP algorithms. The architecture is based on direct execution of data-flow graphs. High computation bandwidth is achieved cost-effectively by exploiting fine-grain parallelism inherent in the target algorithms using simple processor elements called nanoprocessors. The flexible high bandwidth communication network can accommodate a wide range of algorithms, including those with irregular data communication patterns. The distributed control strategy and the data-driven principle of execution result in a highly scalable and modular architecture that can be easily extended to tackle more complex problems. The simplicity and homogeneity of the architecture, the consistent programming paradigm, and the direct architectural support for the execution of data-flow graphs can facilitate development of compiler and synthesis tools needed for rapid prototyping.

A 50 MHz PADDI-2 chip consisting of 48 nanoprocessors, a 2-level communication network, and eight I/O ports has been designed and reported in [116] as a proof of the architectural concept. An application development system based on an assembler, a VHDL simulator, and a VME demonstration/test board was developed to ease the task of programming and mapping practical DSP systems onto the PADDI-2 hardware platform.

The benchmark results demonstrated that the PADDI-2 architecture is flexible and can satisfy the high throughput requirement of a large variety of algorithms. Complex algorithms such as the Hidden Surface Processor and the EPRML Viterbi detector can be implemented using a small number of nanoprocessors.

While we are satisfied with the flexibility and performance of the PADDI-2 architecture, there are many interesting alternatives and improvements worth researching. This dissertation will be concluded by outlining some future research topics:

Enhanced Reconfigurability

Some problems related to the programming of the existing PADDI-2 chip are that programming speed is slow because of the serial scan bus and that the chip can be reconfigured for the next task only after the completion of the current task. By shortening the time needed for reconfiguring the chip and/or allowing reconfiguration of chip to occur concurrently with execution of the current task, the application scope of the architecture can be greatly extended, especially for embedded-controller-type applications. A reconfigurable multiprocessor architecture incorporating these capabilities has recently been proposed in [94].

Compromising SIMD and MIMD architectures

It has been discussed in Section 4.2 that the SIMD control strategy, though efficient in hardware usage, is not scalable and rather constraining in handling algorithms with complex control flow. On the other hand, the MIMD distributed control structure adopted by PADDI-2 is very general and scalable but incurs hardware penalty, e.g., the hardware supported data-driven mechanism. It is interesting to explore compromise architectures that can take advantage of both the simplicity and hardware efficiency of the SIMD architecture and the generality and scalability of the MIMD architecture. An example of such architectures is the XIMD architecture, which is an extension to the VLIW architecture proposed by [109]. However, a very important pre-requisite of this approach is that an efficient compiler for the architecture must also be developed.

On-chip Memories

Many DSP applications can benefit from on-chip memories which relieve I/O bandwidth and reduce memory latency. As discussed in Section 4.5, memory modules can be easily incorporated into the data-driven paradigm of PADDI-2 architecture. A natural and simple improvement of the PADDI-2 chip is to include a few small memory modules on-chip.

Appendix

Appendix.1 48-Nanoprocessor PADDI-2 Chip Pin-List

Table A.1-1 : PADDI-2 Chip Pin-List.

#	Pin Name	Pin Number	#	Pin Name	Pin Number
1	GND	145	2	d0_sb1	144
3	d0_sb2	160	4	d0_weN	193
5	d0[0]	143	6	Vdd	137
7	d0[1]	136	8	d0[2]	176
9	d0[3]	159	10	d0[4]	135
11	GND	129	12	d0[5]	128
13	d0[6]	142	14	d0[7]	127
15	d0[8]	134	16	Vdd	121
17	d0[9]	120	18	d0[10]	126
19	d0[11]	119	20	d0[12]	118
21	GND	113	22	d0[13]	112
23	d0[14]	111	24	d0[15]	110
25	tdi	103	26	Vdd	105
27	tdo	104	28	tms	102

Table A.1-1 : PADDI-2 Chip Pin-List.

#	Pin Name	Pin Number	#	Pin Name	Pin Number
29	ten	94	30	d1[15]	95
31	GND	97	32	d1[14]	96
33	d1[13]	86	34	d1[12]	87
35	d1[11]	77	36	d1[10]	88
37	Vdd	89	38	d1[9]	69
39	d1[8]	78	40	d1[7]	52
41	d1[6]	79	42	GND	80
43	d1[5]	35	44	d1[4]	70
45	d1[3]	18	46	d1[2]	71
47	d1[1]	53	48	Vdd	72
49	d1[0]	36	50	d1_weN	54
51	d1_sb2	19	52	d1_sb1	37
53	GND	55	54	d2_sb1	38
55	d2_sb2	20	56	d2_weN	1
57	d2[0]	21	58	Vdd	56
59	d2[1]	39	60	d2[2]	2
61	d2[3]	3	62	d2[4]	22
63	GND	57	64	d2[5]	40
65	d2[6]	4	66	d2[7]	23
67	d2[8]	5	68	Vdd	58
69	d2[9]	41	70	d2[10]	6

Table A.1-1 : PADDI-2 Chip Pin-List.

#	Pin Name	Pin Number	#	Pin Name	Pin Number
71	d2[11]	24	72	d2[12]	7
73	GND	59	74	d2[13]	42
75	d2[14]	25	76	d2[15]	8
77	Vdd	26	78	Vdd	60
79	GND	43	80	Vdd	9
81	GND	10	82	d3[15]	27
83	GND	61	84	d3[14]	44
85	d3[13]	11	86	d3[12]	28
87	d3[11]	12	88	d3[10]	45
89	Vdd	62	90	d3[9]	13
91	d3[8]	29	92	d3[7]	14
93	d3[6]	46	94	GND	63
95	d3[5]	15	96	d3[4]	30
97	d3[3]	16	98	d3[2]	47
99	d3[1]	31	100	Vdd	64
101	d3[0]	32	102	d3_weN	48
103	d3_sb2	33	104	d3_sb1	49
105	GND	65	106	d4_sb1	66
107	d4_sb2	50	108	d4_weN	17
109	d4[0]	67	110	Vdd	73
111	d4[1]	74	112	d4[2]	34

Table A.1-1 : PADDI-2 Chip Pin-List.

#	Pin Name	Pin Number	#	Pin Name	Pin Number
113	d4[3]	51	114	d4[4]	75
115	GND	82	116	d4[5]	83
117	d4[6]	68	118	d4[7]	84
119	d4[8]	76	120	Vdd	90
121	d4[9]	91	122	d4[10]	85
123	d4[11]	92	124	d4[12]	93
125	GND	98	126	d4[13]	99
127	d4[14]	100	128	d4[15]	101
129	Reserve	108	130	Vdd	106
131	ck	107	132	resetN	109
133	gstall	117	134	d5[15]	116
135	GND	114	136	d5[14]	115
137	d5[13]	125	138	d5[12]	124
139	d5[11]	133	140	d5[10]	123
141	Vdd	122	142	d5[9]	141
143	d5[8]	132	144	d5[7]	158
145	d5[6]	131	146	GND	130
147	d5[5]	175	148	d5[4]	140
149	d5[3]	192	150	d5[2]	139
151	d5[1]	157	152	Vdd	138
153	d5[0]	174	154	d5_weN	156

Table A.1-1 : PADDI-2 Chip Pin-List.

#	Pin Name	Pin Number	#	Pin Name	Pin Number
155	d5_sb2	191	156	d5_sb1	173
157	GND	155	158	d6_sb1	172
159	d6_sb2	190	160	d6_weN	209
161	d6[0]	189	162	Vdd	154
163	d6[1]	171	164	d6[2]	208
165	d6[3]	207	166	d6[4]	188
167	GND	153	168	d6[5]	170
169	d6[6]	206	170	d6[7]	187
171	d6[8]	205	172	Vdd	152
173	d6[9]	169	174	d6[10]	204
175	d6[11]	186	176	d6[12]	203
177	GND	151	178	d6[13]	168
179	d6[14]	185	180	d6[15]	202
181	Vdd	184	182	Vdd	150
183	GND	167	184	Vdd	201
185	GND	200	186	d7[15]	183
187	GND	149	188	d7[14]	166
189	d7[13]	199	190	d7[12]	182
191	d7[11]	198	192	d7[10]	165
193	Vdd	148	194	d7[9]	197
195	d7[8]	181	196	d7[7]	196

Table A.1-1 : PADDI-2 Chip Pin-List.

#	Pin Name	Pin Number	#	Pin Name	Pin Number
197	d7[6]	164	198	GND	147
199	d7[5]	195	200	d7[4]	180
201	d7[3]	194	202	d7[2]	163
203	d7[1]	179	204	Vdd	146
205	d7[0]	178	206	d7_weN	162
207	d7_sb2	177	208	d7_sb1	161

Appendix.2 PADDI-2 Instruction Mnemonics

Table A.2-1 : PADDI-2 Instructions.

#	Mnemonics[a]	Instruction	Result[b]
1	add(src1, src2)	2's complement addition	src1 + src2
2	addcc(src1, src2, cc)	2's complement addition w/ cc	src1 + src2 + cc
3	ads(src1, n)	arithmetic down-shift	src1 >> n, n=0,1,2
4	and(src1, src2)	logical AND	src1 & src2
5	aus(src1, n)	arithmetic up-shift	src1 << n, n=2
6	inv(src1)	logical inverse	~src1
7	mstart(src1, src2)	2nd-order modified booth multiplication step (zero partial sum)	booth(src1, src2)
8	mstep(src1, src2, src3)	2nd-order modified booth multiplication step	booth(src1, src2) + src3
9	or(src1, src2)	logical OR	src1 | src2
10	selcc(src1, src2)	select on cc	cc? src1 : src2
11	sub(src1, src2)	2's complement subtract	src1 - src2
12	subcc(src1, src2, cc)	2's complement subtract w/ cc	src1 + ~src2 + cc
13	xor(src1, src2)	logical exclusive-OR	src1 ^ src2

a. srcN = q0/q0s/q1/q1s/q2/q2s/R1-R6, where qI means the datum is read and then discarded from queue and qIs means the datum is read but still kept in queue.
b. In C-Language syntax.

Appendix.3 Board Controller Registers

Table A.3-1 : Board Controller Registers

Register Number	Register Name	Access Type
0	Control Register	Read/Write
1	Scan/Data-Out Register	Read/Write
2	Data-In Register	Read Only

Table A.3-2 : Board Control Register

Bit #	Name
15	ST (Stop)
14	SS (Single-Step)
13	CU (Capture/Update)
12	SE (Scan-Enable)
11	SF (Scan-reg Full)
10	DF (Data-In reg Full)
9-4	Reserved
3-0	CS[3:0][a] (Chip-Select [3:0])

a. The 4 Chip-Select bits allow one board controller chip to support up to 16 PADDI-2 chips.

Appendix.4 Layout of PADDI-2 VME Board

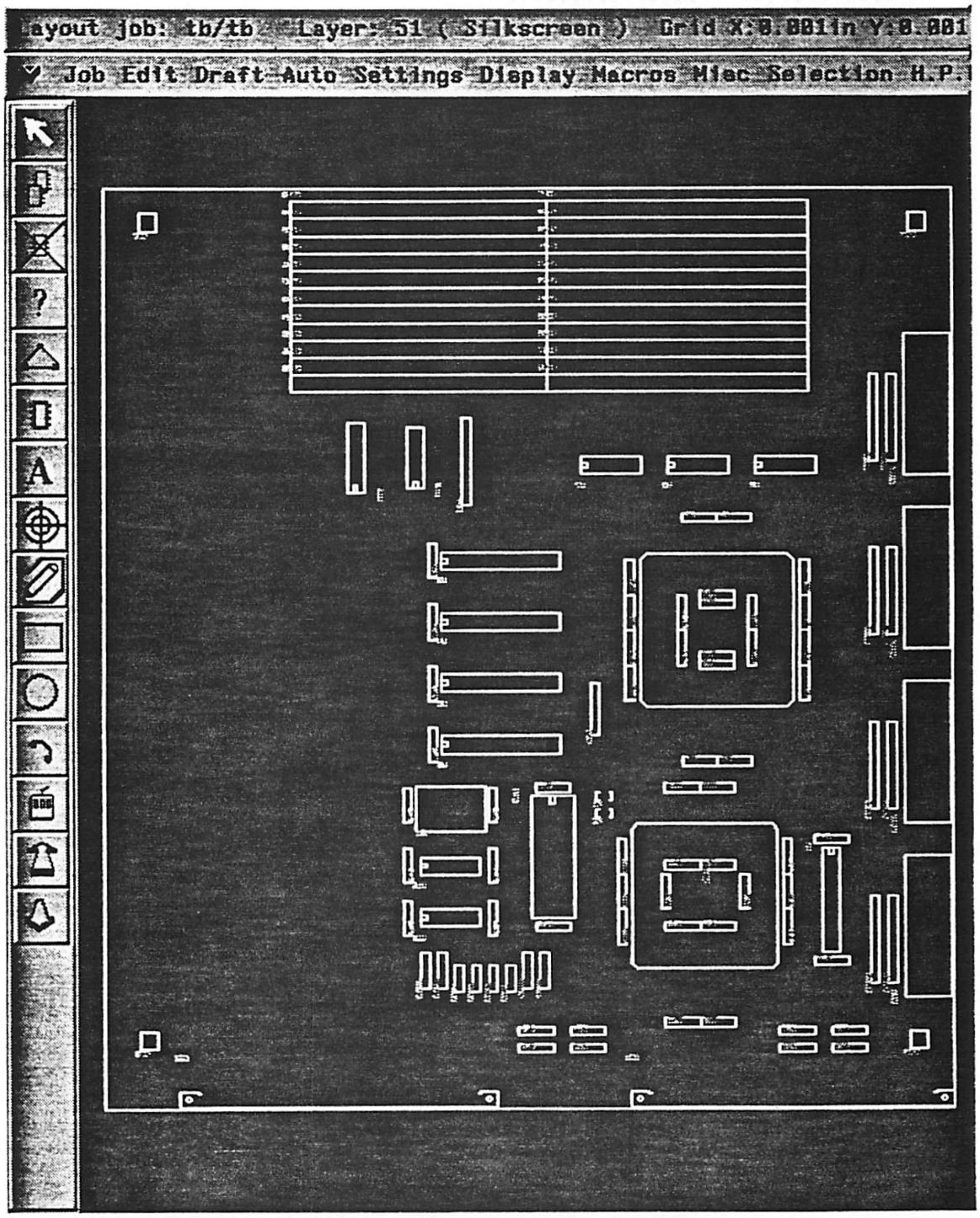

ayout job: tb/tb   Layer: 51 (Silkscreen)   Grid X:0.001in Y:0.001
Job Edit Draft Auto Settings Display Macros Misc Selection H.P.
?
A

Figure A-1 : Layout of PADDI-2 VME Board.

Appendix.5 Nanoprocessor Assembler Programs for Hidden Surface Processors

```
/* ****************************** */
/* XPROC.s notes:                 */
/*                                */
/*  1) X, DX comes in on q0       */

.init q0 = q
.init q1 = r
.init q2 = r
.init r6 = 1
.init pc = 0

0:
r0 = sub(q0, r6),
c0 = ra                       /* output new X */
cc0 = s,
next = 1;

1:
r3 = sub(q0, r6),             /* store value of DX-1 in r3 */
cc1 = s,
next = cc0? 3:2

2:
r0 = add(r3, r6),             /* output the original DX */
c0 = ra,
fo_s = 0,                     /* de-assert enable */
next = 0;

3:                            /* has reached beginning of segment */
r0 = ads(r3, 0),
c0 = ra,                      /* output new DX */
next = cc1? 5:4;

4:                            /* within line segment */
fo_s = 1,                     /* assert enable */
next = 0;

5:                            /* no longer within line segment */
fo_s = 0,                     /* de-assert enable */
next = 0;
```

```
/* ****************************** */
/* ZPROC.s notes                  */
/*                                */
/*   1) Z, reset come in on q0    */
/*   2) DZ is always implemented as constant     */
/*   3) flag from XPROC comes in on f0            */
/*   4) r0 used as a scrap register */
/*   5) r6 used as Zmin           */

.init q0 = q, q1 = r, q2 = r, pc = 3
.init r6 = 10000               /* set Zmin to be a default distance */
.init r5 = 10000               /* this is the reset value for Zmin */
.init r4 = 3                   /* constant DZ value */

3:
r0 = sub(q0s,r6),              /* determine in advance if need to update Zmin */
cc0 = s,                       /* s = 1 means update; s = 0 means don't update */
next = f0? 1:0;                /* test if within segment */

0:
r0 = ads(q0,0),                /* not within segment, so: */
c0 = ra,                       /* pass original Z */
cc0 = s, next=2;               /* see if Z was -; if so, need to reset Zmin */

2:
next = cc0? 7:6;               /* check if need to reset Zmin */

7:
r6 = ads(r5, 0),               /* reset asserted, so set r6 to be Zmin */
next = 3;                      /* also note that a reset will not put out a fo token    */

1:
r0 = add(q0s,r4),              /* pass Z+dZ */
c0 = ra, next = cc0? 5:4;

5:
r6 = ads(q0,0), next=6;        /* update Zmin */

4:
r0 = ads(q0,0), next=6;         /* flush q0 queue */

6:
fo_s = cc0, next=3;            /* set intensity flag to proper value */
```

```
/* ****************************** */
/* IPROC.s notes:                 */
/*                                */
/*  1) I comes in on q0           */
/*  2) SWAP comes in on f0        */
/*  3) EOF comes in on f1, leaves on fo */
/*  3) r6 used to keep Imax       */

.init q0 = q
.init q1 = r
.init q2 = r

.init r6 = 0                  /* set Imin to be background intensity */
.init r5 = 0                  /* this is the reset value for back. intens. */
.init pc = 0

0:
r3 = ads(q0, 0),              /* check if Iin < 0 , ie FLUSH */
c0 = ra,                      /* output both Iout, FLUSH */
cc0 = s,
next = 2;

2:
next = cc0? 5:4;              /* branch on FLUSH */

4:
next = f0? 1:0;               /* branch on SWAP */

1:
r6 = ads(r3, 0),              /* replace Imax */
next = 0;

5:                            /* flush cycle for the 1st pixel processor */
r0 = ads(r6, 0),
c0 = ra,                      /* send own Imax */
fo = 1,                       /* output EOF */
next = 6;

6:
r6 = ads(r5,0),               /* reset r6 */
next = 0;
```

Appendix.6 Nanoprocessor Assembler Programs for Viterbi Detector

```
/* ****************************** */
/* P0.s note                      */
/*                                */
/*  1) Y comes in on q0           */
/*  2) S0 comes in on q1          */
/*  3) This is program for state 0,5,6 ACS      */
/*  4) So need to calculate BM(-1) = 0.5+Y      */
/*  5) BM(0) = 0                  */
/*  6) r6 used as a scrap register  */

.init q0 = q, q1 = q, q2 = r
.init r5 = 4                      /* represent 0.5 with 3 decimals */
.init pc = 0

0:
r0 = add(r5,q0),                  /* calculate BM(-1) = 0.5+Y */
c0 = ra,                          /* send it to the P1 processor */
next = 1;

1:
r0 = ads(q1,0),                   /* calculate BM(0)+S0 */
c0 = ra,
next = 0;



/* ****************************** */
/* P1.s notes                     */
/*                                */
/*  1) BM(+1), then BM(+2)+S0 comes in on q0*/
/*  2) S1 comes in on q1           */

.init q0 = q, q1 = q, q2 = r
.init r6 = 0, pc = 0

0:
next = f0? 3:2;                   /* if reset, don't absorb Q values */

2:                                /* no reset */
r5 = add(q0,q1),                  /* calculate BM(+1)+S1 */
```

```
next = 4;

4:
r0 = sub(q0s,r5),                  /* compare BM(+2)+S0 and BM(+1)+S1 */
cc0 = s,
next = 5;

5:
r0 = sel(q0, r5, cc0),             /* if cc0=0, r0=q0, else r0=r5 */
c0 = ra,                           /* check this! */
fo_s = cc0,                        /* set output flag */
next = 0;

3:                                 /* reset enabled */
r0 = ads(r6,0),
c0 = ra,                           /* output 0 */
next = 0;



/* ****************************** */
/* RE.s notes:                    */
/*  1) Inputs come in on q0,q1    */
/*  2) Decision comes in on f0, reset on f1     */
/*  3) Output is on c0            */
/*  4) Decode output is on fo     */

.init q0 = q, q1 = q, q2 = r
.init r5 = 0, r6 = 1, pc = 0

0:
next = f1? 3:2;                    /* if reset, don't absorb Q values */

2:
next = f0? 5:4;                    /* branch based on decison */

4:                                 /* decision = 0 */
r0 = aus(q0),                      /* shift appropriate input left by 1 bit */
cc0 = s,
c0 = ra,
next = 6;

6:
```

```
r0 = ads(q1,0),                 /* flush q1 queue */
fo_s = cc0,
next = 0;

5:                              /* decision = 1 */
r0 = aus(q1),                   /* shift appropriate input left by 1 bit */
next = 7;

7:
r0 = add(ra,r6),                /* place decision bit into LSB */
cc0 = s,
c0 = ra,                        /* so output is input shifted left by */
next = 1;                       /* one bit, with the decision occupying the LSB */

1:
r0 = ads(q0,0),                 /* flush the q0 queue */
fo_s = cc0,
next = 0;

3:                              /* reset enabled */
r0 = ads(r5,0),                 /* output 0 */
c0 = ra,
next = 0;
```

Bibliography

[1] Arvind and D. Culler, "Dataflow Architectures," *Annual Reviews in Computer Science*, MIT, Laboratory for Computer Science, 1986.

[2] Avenhaus, E., "On the design of digital filters with coefficients of limited word length," *IEEE Trans. on Audio and Electroacoustics*, vol. 20, pp. 206-212, 1972.

[3] Baccarani, G., M. Wordeman and R. Dennard, "Generalized Scaling Theory and its Application to 1/4 Micrometer MOSFET Design," *IEEE Transactions on Electron Devices*, ED-31(4), p. 452, 1984.

[4] Bakoglu, H., *Circuits, Interconnections, and Packaging for VLSI*, Addison-Wesley, Menlo Park, CA, 1990.

[5] Berkhout, P. and L. Eggermont, "Digital Audio Systems," *IEEE ASSP Magazine*, pp. 86-99, May 1989.

[6] Bier, J., S. Sriram and E. Lee, "A Class of Multiprocessor Architectures for Real-Time DSP," *VLSI Signal Processing IV*, Nov. 1990.

[7] Bisiani, R., "System Implementation Strategies for Speech," *Speech and Natural Language Workshop*, June 1990.

[8] Bisiani, R., et al., "BEAM: An Accelerator for Speech Recognition," *International Conference on Acoustics, Speech and Signal Processing*, June 1989.

[9] Black, P. and T. Meng, "A 140 Mb/s, 32-state, Radix-4 Viterbi Decoder," *IEEE J. Solid-State Circuits*, Vol. 27, pp. 1877-1885, Dec 1992.

[10] Black, P. and T. Meng, "A 1 Gb/s, 4-state, Sliding Block Viterbi Decoder," *Symp. on VLSI Circuits*, Jan 1993.

[11] Black, P., "Algorithms and Architectures for High Speed Viterbi Decoding," PhD thesis, Stanford University, CA, March 1993.

[12] Blanz, W. E., et al., "Algorithms and Architectures for Machine Vision," *Handbook of Signal Processing*, Marcell Decker, 1988.

[13] Borkar, S., et al., "iWarp: An Integrated Solution to High-Speed Parallel Computing," *Proceedings Supercomputing*, Nov 1988.

[14] Brodersen, R. W., and J. Rabaey, "Evolution of Microsystem Design," *ESSCIRC'89: Proceedings of the 15th European Solid State Circuits Conference*, Sept. 1989.

[15] Brodersen, R. W., et al., "Technologies for Personal Communications," *Proceedings of the VLSI Symposium '91 Conference*, Japan, pp. 5-9, 1991.

[16] Brown, C., "The Programmable-Logic Assault," *Electronic Engineering Times*, Feb. 1993.

[17] Brown, S., "An Overview of Technology, Architecture and CAD Tools for Programmable Logic Devices," *IEEE Custom Integrated Circuits Conference*, 1994.

[18] Brunvand, E., "Using FPGA's to implement self-timed systems," *Journal of VLSI Signal Processing*, vol. 6, 1993.

[19] Chandrakasan, A., A. Burstein and R. Brodersen, "A Low Power Chipset for Portable Multimedia Applications," *Proceedings of the International Solid-State Circuits Conference '94*, San Francisco, CA, pp. 82-83, February 1994.

[20] Chandrakasan, A., R. Allmon, A. Stratakos, and R. W. Brodersen, "Design of Portable Systems," *Proceedings of CICC '94*, San Diego, May 1994.

[21] Chandrakasan, A., S. Sheng, and R. W. Brodersen, "Low-Power Techniques for Portable Real-Time DSP Applications," *VLSI Design '92*, India, pp. 203-208, 1992.

[22] Chandrakasan, A., S. Sheng, and R. W. Brodersen, "Low-Power CMOS Design," *IEEE Journal of Solid-State Circuits*, pp. 472-484, April 1992.

[23] Chandrakasan, A., M. Potkonjak, J. Rabaey, and R. W. Brodersen, "Optimizing Power Using Transformations," submitted to *IEEE Transactions on Computer-Aided Design*, May 1993.

[24] Chen, D. K., Hong-Men Su, and Pen-Chung Yew, "The Impact of Synchronization and Granularity on Parallel Systems," *International Symposium on Computer Architecture*, 1990.

[25] Chen, D., L. Guerra, E. Ng, D. Schultz, and J. Rabaey, "A Field-Programmable Architecture for High Speed Digital Signal Processing Application," *ACM International Workshop on Field-Programmable Gate Arrays*, 1992.

[26] Chen, D., and J. Rabaey, "A Reconfigurable Multiprocessor IC for Rapid Prototyping of Real-Time Data Paths", *IEEE International Solid-State Circuits Conference*, Feb. 1992.

[27] Chen, D., "Programmable Arithmetic Devices for High Speed Digital Signal Processing", PhD thesis, University of California at Berkeley, May, 1992.

[28] Dennis., J. B., "First Version Data Flow Procedure Language", Technical Memo MAC TM61, May 1975, MIT Laboratory for Computer Science.

[29] De Man, H., F. Catthoor, G. Goossens, J. Vanhoof, J. Van Meerbergen, S. Note, and J. Huisken, "Architecture-Driven Synthesis Techniques for Mapping Digital Signal Processing Algorithms into Silicon," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 319-335, Feb. 1990.

[30] Devaney, P. of Matsushita Applied Research Laboratory, *Personal Communication*, Feb., 1994.

[31] Duncan, P., et al., "Hi-PASS: A Computer-Aided Synthesis System for Maximally Parallel Digital Signal Processing ASIC's," *International Conference on Acoustics, Speech and Signal Processing*, Mar. 1992.

[32] Engels, M., R. Lauwereins, and J. Peperstraete, "Rapid Prototyping for DSP Systems with Multiprocessors," *IEEE Design & Test of Computers*, June 1991.

[33] Ernst, R., "Long Pipelines in Single-Chip Digital Signal Processors: Concepts and Case Study," *IEEE Transactions on Circuits and Systems*, Jan. 1991.

[34] Fellman, R., "Design Issues and an Architecture for the Monolithic Implementation of a Parallel Digital Signal Processor," *IEEE Transactions on Acoustics, Speech and Signal Processing*, May 1990.

[35] Flynn, M. J., "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, Sept. 1972.

[36] Forney, G. D. Jr., "The Viterbi algorithm," *Proc. IEEE*, vol. 61, no. 3, pp. 268-278, March 1973.

[37] Fortes, J. A. B., and B. W. Wah, eds., "Special Issue on Systolic Arrays - From Concept to Implementation," *Computer*, July 1987.

[38] Fountain, T. J., et al., "The CLIP7A Image Processor," *IEEE Transactions on Pattern Analysis & Machine Intelligence*, May, 1988.

[39] Freeman, R., "User-programmable Gate Arrays," *IEEE Spectrum*, Dec., 1988.

[40] Fukushima, T, "A Survey of Image Processing LSI's in Japan," *IEEE International Conference on Pattern Recognition*, 1990.

[41] Geurts, W. and F. Catthoor, "DSP Applications suited for Lowly Multiplexed Architectures", *ASICS Open Workshop on Synthesis Techniques for (Lowly) Multiplexed Datapaths*, Aug. 1990.

[42] Gharavi, H., P. Pirsch, and H. Yasuda, "Special Issue on VLSI Implementation For Digital Image and Video Processing Applications," *IEEE Transactions on Circuits and Systems*, pp. 1259-1365, Oct. 1989.

[43] Gonzalez, R. and R. Woods, *Digital Image Processing*, Addison-Wesley Publishing Company, Inc., 1993.

[44] Goodman, D., "Trends in Cellular and Cordless Communications," *IEEE Communications Magazine*, pp. 31-40, June 1991.

[45] Goto, Junichi, et. al., "A 250 MHz 16b 1-Million Transistor BiCMOS Super-High-Speed Video Signal Processor," *International Solid-State Circuits Conference*, 1991.

[46] He, S. and M. Torkelson, "FPGA Application in a SBus-based Rapid Prototyping System for AS-DSP," *International Workshop on Field-Programmable Logic and Applications*, Aug. 1992.

[47] He, S. and M. Torkelson, "FPGA Implementation of FIR Filters Using Pipelined

Bit-Serial Canonical Signed Digit Multipliers," *IEEE Custom Integrated Circuits Conference*, 1994.

[48] Heurikon Corporation, 8310 Excelsior Drive, Madison, WI 53717.

[49] Hilfinger, P., "A High-Level Language and Silicon Compiler for Digital Signal Processing," *Proceedings of Custom Integrated Circuits Conference*, Los Alamitos, CA, pp. 213-216, 1985.

[50] Hoang, P. and J. M. Rabaey, "MsDAS: A Compiler for Multiprocessor DSP Implementation," *International Conference on Acoustics, Speech and Signal Processing*, Mar. 1992.

[51] Homewood, M., et al., "The IMS T800 Transputer," *IEEE Micro*, Oct. 1987.

[52] Johnson, H. W. and M. Graham, *High-Speed Digital Design*, PTR Prentice-Hall, Inc., 1993.

[53] Jurgen, R. K., "The Challenges of Digital HDTV", *IEEE Spectrum*, April, 1991.

[54] *IEEE P1149.1 JTAG Serial Scan Interface Specification*, 1990.

[55] Keutzer, K., "Three Competing Design Methodologies for ASIC's: Architectural Synthesis, Logic Synthesis and Module Generation," *ACM/IEEE Design Automation Conference*, Feb. 1989.

[56] Koh, W., A. Yeung, P. Hoang and J. Rabaey, "A Configurable Multiprocessor System for DSP Behavioral Simulation," *IEEE International Symposium on Circuits and Systems*, 1989.

[57] Koren, I., et al., "A Data-Driven VLSI Array for Arbitrary Algorithms," *Com-*

puter, Oct. 1988.

[58] Kung, S. Y., et al., "Wavefront Array Processors - Concept to Implementation," *Computer*, July 1987.

[59] Labrousse, J., et al., "A 50 MHz Microprocessor with a VLIW Architecture", *ISSCC*, 1990.

[60] Leacock, T., et al., "HDTV Digital Camera Processor," *135th SMPTE Technical Conference*, Oct., 1993.

[61] Lee, B. W., et al., "Data Flow Processor for Multi-standard Video Codec," *IEEE Custom Integrated Circuits Conference*, 1994.

[62] Lee, E. A., "Programmable DSP Architectures: Part I," *IEEE ASSP Magazine*, Oct. 1988.

[63] Lee, E. A., "Consistency in Dataflow Graphs," *IEEE Transactions on Parallel and Distributed Systems*, April, 1991.

[64] Lipsett, R., C. Schaefer and C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, 1989.

[65] Little, M. J., et al., "3-D Computer for Advanced Fire Control," *First Annual Fire Control Symposium*, Oct. 1990.

[66] LSI Logic, Inc., *Data sheet: L64853 SBus DMA Controller*, June, 1990.

[67] Mao, W. and S. Y. Kung, "A Real Time Pipeline Interleaving approach for Image/ Video Signal Processing", *ICASSP*, pp. 3046-3049, 1990.

[68] McNally, G. W., "Digital Audio in Broadcasting," *IEEE ASSP Magazine*, pp 26-

44, Oct. 1985.

[69] Meier, S, S. Perissakis, and J. Wawrzynek, "High-Speed CMOS Signaling," *Research Summary*, EECS/ERL, University of California at Berkeley, 1994.

[70] Mello, J. and P. Wayner, "Wireless Mobile Communications," *Byte*, vol. 18, no. 2, pp. 146-153, Feb. 1993.

[71] Mendelson, B., and G. M. Silberman, "Mapping Data Flow Programs on a VLSI Array of Processors," *International Symposium on Computer Architecture*, 1987.

[72] Minami, T., et al., "A 300 MOPS Video Signal Processor with a Parallel Architecture," *Journal of Solid-State Circuits*, Dec. 1991.

[73] Oppenheim, A. V., and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice Hall, NJ, 1989.

[74] Papamichalis, P., *Digital Signal Processing Applications with the TMS320 Family, Vol. 3*, Prentice Hall, 1989.

[75] Parhi, K. and D. Messerschmitt, "Pipeline Interleaving and Parallelism in Recursive Digital Filters - Part I: Pipelining using Scattered Look-ahead and Decomposition," *IEEE Transaction on Acoustics, Speech, and Signal Processing*, vol. 37, July 1989.

[76] PLX Technology, Inc., *Data Sheet: VMEbus Slave Module Interface Device, VME 2000*, 1989.

[77] Potkonjak, M. and J. Rabaey, "Exploring the Algorithmic Design Space using High Level Synthesis," *VLSI Signal Processing VI*, pp. 123-131, 1993.

[78] Putname, L. K., P. Lucht and J. Davis, "A High-Speed Architecture for Image Computation," *SMPTE*, June 1988.

[79] Newman, W. M. ,and R. Sproull, *Principles of Interactive Computer Graphics*, Second Edition, McGraw Hill, NY, 1979.

[80] Nielsen, R. O., *Sonar Signal Processing*, Artech House Inc., 1991.

[81] Nishizawa, T., et al., "A Hidden Surface Processor for 3-Dimension Graphics," *International Solid-State Circuit Conference*, Feb. 1988.

[82] Rabaey, J. M., et al., "A Large Vocabulary Real-Time Continuous Speech Recognition System," *VLSI Signal Processing III*, pp. 62-74, IEEE Press, Nov., 1989.

[83] Rabaey, J. M., C. Chu, P. Hoang, and M. Potkonjak, "Fast Prototyping of Datapath-Intensive Architectures," *IEEE Design & Test of Computers*, pp. 40-51, June 1991.

[84] Rao, S. K., "Regular iterative algorithms and their implementation on processor arrays," Ph.D. thesis, Stanford University, 1985.

[85] Roe, D., A. Gorin and P. Ramesh, "Incorporating Syntax into the Level-Building Algorithm on a Tree-Structured Parallel Computer," *IEEE Transaction on Acoustics, Speech, and Signal Processing*, 1989.

[86] Ruetz, P. and R. Brodersen, "A Real-time Image Processing Chip Set," *International Solid-State Circuit Conference*, Feb. 1986.

[87] Sandbank, C. P. ed., *Digital Television*, John Wiley and Sons, 1990.

[88] Schmidt, U., "Data-Wave: a Data Driven Video Signal Array Processor," *Hot*

Chips II: A Symposium on High Performance Chips, Aug. 1990.

[89] Schmidt, U., and S. Mehgardt, "Wavefront Array Processor for Video Applications," *International Conference on Computer Design*, 1990.

[90] Senn, P., et al., "A 27 MHz Digital-to-Analog Video Processor," *International Solid-State Circuit Conference*, Feb. 1988.

[91] Skillicorn, D. B., "A Taxonomy for Computer Architectures," *IEEE Computer*, Nov. 1988.

[92] Sluyter, R. J., et al., "A Programmable Video Signal Processor," *ICASSP*, 1989, Glaagow, U. K..

[93] SPARC Technology Business, "Industry's First Processor to offer On-chip Processing Power for Real-time Multimedia Compression and Decompression," Announcement, Sept. 19, 1994.

[94] Srini, V. P., "Low-Power Parallel DSP Chip for Multimedia Data and 3-D Image Processing," *Small Business Technology Transfer Program Proposal* (AF 94T0001), U.S. Department of Defense, 1994.

[95] Sugawara, T., et al., "Viterbi Detector including PRML and EPRML," *IEEE Transactions on Magnetics*, Vol. 29, Nov. 1993.

[96] Sun, J. S., M. B. Srivastava, and R. W. Brodersen, "SIERA: A CAD Environment for Real-Time Systems," *Third IEEE/ACM Physical Design Workshop on Module Generation and Silicon Compilation*, May 1991.

[97] Sun Microsystems, Inc., *Sbus Application Notes*, July, 1990.

[98] Synopsys, Inc., *VHDL System Simulator Core Programs Manual*, Dec. 1992.

[99] Tewksbury, S. K. and L. A. Hornak, "Wafer Level System Integration: A Review," *IEEE Circuits and Devices Magazine*, Sept. 1989.

[100] Thiele, L., "On the hierarchical design of VLSI processor arrays," *IEEE Symp. on Circuits and Systems*, 1988.

[101] Toyokura, M., et al., "A Video Digital Signal Processor with a Vector-Pipeline Architecture", *ISSCC*, 1992.

[102] Veendrick, H., et al., "A 1.5 GIPS Video Signal Processor (VSP)," *IEEE Custom Integrated Circuits Conference*, 1994.

[103] Viewlogic Systems, Inc., *Workview User's Guide*, 1991.

[104] *The VMEbus Specification C.1*, Printex Publishing, 1984.

[105] Volkers, H., et al., "Cache Memory Design for the Data Transport to Array Processors," *ICASSP*, 1990.

[106] Ward, J., et al., "Figures of Merit for VLSI Implementations of Digital Signal Processing Algorithms," *Proceedings of the Institute of Electrical Engineers*, vol. 131, Part F, pp. 64-70, February 1984.

[107] Wawrznyek, J., "A Reconfigurable Concurrent VLSI Architecture for Sound Synthesis," *VLSI Signal Processing II*, Nov. 1986.

[108] Wind River Systems, Inc., *VxWorks Reference Manual*, 1989.

[109] Wolfe, A and J. P. Shen, "A Variable Instruction Stream Extension to the VLIW Architecture," *ASPLOS*, 1991.

[110] Xilinx Inc., *The Programmable Logic Data Book*, 1994.

[111] Yamashita, N., et al., "A 3.84GIPS Integrated Memory Array Processor LSI with 64 Processing Elements and 2 Mb SRAM", *IEEE International Solid-State Circuits Conference*, Feb. 1994.

[112] Yang, K., et al., "A Flexible Motion-Vector Estimation Chip for Real-time Video Codecs," *IEEE Custom Integrated Circuits Conference*, 1990.

[113] Yeung, A. K. and J. Rabaey, "A Data-driven Architecture for Rapid Prototyping of High Throughput DSP Algorithms," *IEEE VLSI Signal Processing Workshop*, Oct. 1992.

[114] Yeung, A. K. and J. Rabaey, "A Reconfigurable Data-driven Multiprocessor Architecture for Rapid Prototyping of High Throughput DSP Algorithms," *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, 1993.

[115] Yeung, A. K. and J. Rabaey, "A 210 Mb/s Radix-4 Bit-Level Pipelined Viterbi Decoder," *International Solid-State Circuit Conference*, Feb. 1995.

[116] Yeung, A. K. and J. Rabaey, "A 2.4 GOPS Reconfigurable Data-Driven Multiprocessor IC for DSP," *International Solid-State Circuit Conference*, Feb. 1995.