LOW-POWER DOMAIN-SPECIFIC PROCESSORS FOR DIGITAL SIGNAL PROCESSING

by

Arthur Abnous

Memorandum No. UCB/ERL M01/16

6 April 2001

LOW-POWER DOMAIN-SPECIFIC PROCESSORS FOR DIGITAL SIGNAL PROCESSING

by

Arthur Abnous

Memorandum No. UCB/ERL M01/16

6 April 2001

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Low-Power Domain-Specific Processors for Digital Signal Processing

by

Arthur Abnous

B.S. (University of California, Irvine) 1989
M.S. (University of California, Irvine) 1991

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy
in

Engineering - Electrical Engineering
and Computer Sciences

in the

GRADUATE DIVISION

of the

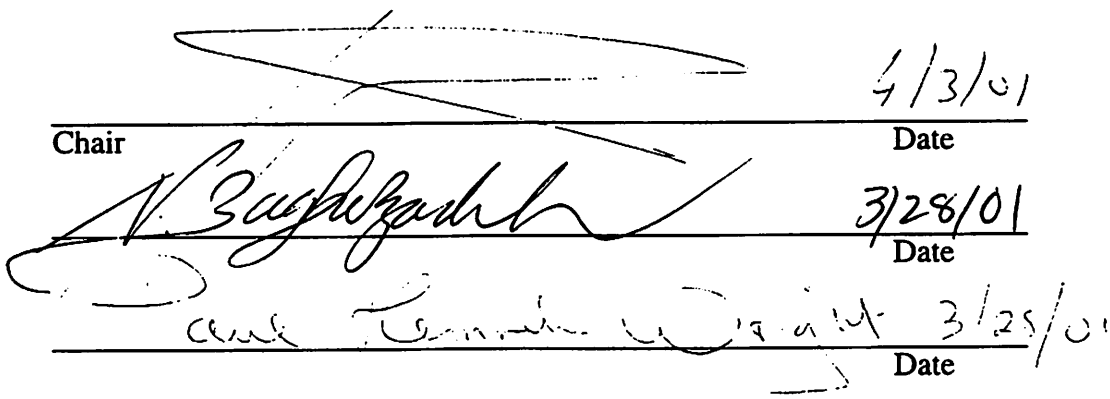
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Jan M. Rabaey, Chair
Professor Nader Bagherzadeh
Professor Paul K. Wright

Spring 2001

The dissertation of Arthur Abnous is approved:



Chair — Date 4/3/01

Date 3/28/01

Date 3/25/01

University of California, Berkeley

Spring 2001

Low-Power Domain-Specific Processors for Digital Signal Processing

Abstract

Low-Power Domain-Specific Processors for Digital Signal Processing

by

Arthur Abnous

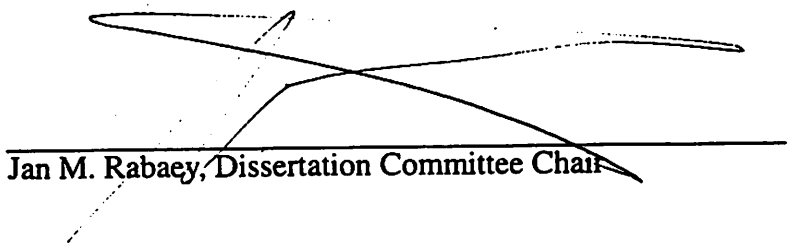
Doctor of Philosophy in Engineering - Electrical Engineering
and Computer Sciences

University of California, Berkeley

Professor Jan M. Rabaey, Chair

Rapid advances in portable computing and communication devices require implementations that must not only be highly energy efficient, but they must also be flexible enough to support a variety of multimedia services and communication capabilities. The required flexibility dictates the use of programmable processors in implementing the increasingly sophisticated digital signal processing algorithms that are widely used in portable multimedia terminals. However, compared to custom, application-specific solutions, programmable processors often incur significant penalties in energy efficiency and performance. The approach taken in this work was to explore ways of trading off flexibility for increased efficiency. This approach was based on the observation that for a given domain of signal processing algorithms, the underlying computational kernels that account for a large fraction of execution time and energy are very similar. By executing the dominant kernels of a given domain of algorithms on dedicated, optimized processing elements that can execute those kernels with a minimum of energy overhead, significant energy savings can potentially be achieved. Thus, the approach taken in this work yields processors that are domain-specific. The main contribution of this work is a reusable architecture tem-

plate, named Pleiades, that can be used to implement domain-specific, programmable processors for digital signal processing algorithms. The Pleiades architecture template relies on a heterogeneous network of processing elements, optimized for a given domain of algorithms, that can be reconfigured at run time to execute the dominant kernels of the given domain. To verify the effectiveness of the Pleiades architecture, prototype processors were designed, fabricated, and evaluated. Measured results and benchmark studies demonstrate the effectiveness of the Pleiades architecture.



Jan M. Rabaey, Dissertation Committee Chair

To my parents,

Herand Abnous and Anoush Mekaili

Table of Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, I would like to thank my research advisor, Professor Jan Rabaey. It has been an honor, a privilege, and a pleasure to work and learn under his guidance. I am very grateful for his advice, encouragement, support, and patience. The Pleiades project would not have been possible without his vision and leadership. I will always be grateful to him for the very many things, both technical and non-technical, that I have learned from him.

I would like to thank Professors John Wawrzynek, Paul Gray, and Arie Segev for serving on my qualifying examination committee and for their feedback and advice on my research. I would like to thank Professors Nader Bagherzadeh and Paul Wright for serving on my dissertation committee and reviewing this dissertation. I would also like to thank Professor Bob Brodersen. He and Jan provided a stimulating and enjoyable research and learning environment, for which I am very grateful.

A number of people have made important contributions to the Pleiades project, and I would like to thank them for their contributions and for their help. Many thanks go to

Katsunori Seno and Yuji Ichikawa for their invaluable contributions to the design and implementation of the P1 prototype and the P1 benchmark study. Vandana Prabhu designed the P1 test board, helped in testing the P1 prototype, and made many contributions to the design and implementation of the Maia processor. Varghese George helped with the design of the P1 prototype and the Maia processor. To the Maia design he contributed the FPGA architecture that he had developed in his Ph.D. research. Marlene Wan helped with the P1 prototype, the P1 benchmark study, and the Maia processor. She was also responsible for developing the Pleiades design methodology. Suet-Fei Li also contributed to the development of the Pleiades design methodology. Martin Benes designed the asynchronous handshake circuits of the Maia processor and contributed to the design and implementation of the Maia processor. Hui Zhang designed the communication network of the Maia processor. He also led the implementation effort for the Maia processor. Erik Kusse provided benchmark data for the Xilinx FPGA devices. I thank these friends for their many contributions. I am grateful for the privilege of working with them and learning from them.

I am grateful for the privilege and pleasure of learning from and enjoying the friendship of a number of individuals at Berkeley. They include Arya Behzad, Alfred Yeung, Paul Landman, Sam Sheng, Anantha Chandrakasan, Andy Burstein, Tom Burd, Lisa Guerra, Renu Mehra, Ole Bentz, Shankar Narayanaswamy, Andy Abo, Arnold Feldman, Anthony Stratakos, David Lidsky, Randy Allmon, Richard Edell, Roy Sutton, Vason Srini, Rhett Davis, Dennis Yee, Ian O'Donnell, Peggy Laramie, Trevor Pering, Jeff Gilbert, Tom Truman, Engling Yeo, My Le, Srenik Mehta, Chris Rudell, Jeff Weldon, Sekhar Narayanaswami, and Keith Onodera.

I am grateful for the administrative assistance provided by Ruth Gjerde, Heather Brown, Tom Boot, Peggye Browne, Elise Mills, and Carol Sitea. I am also grateful for the

technical support provided by Brian Richards and Susan Mellers. Special thanks go to Kevin Zimmerman for his diligent and cheerful assistance with all matters related to computing resources.

I wrote this dissertation while working for Mehdi Hatamian at Broadcom Corporation. I would like to thank him for his support and encouragement in completing this dissertation.

On the personal side, I would like to thank my parents, Herand Abnous and Anoush Mekaili, and my sisters, Stella and Elena, for their unconditional love, support, and encouragement. I could not have done it without them. I would like to thank my cousin, Razmik Abnous, his wife, Suzanne, and their boys, Sevan, Haig and Armen, for their love and support. I would also like to thank Karen Bradford for her friendship and support.

CHAPTER 1

Introduction

An important trend that has been a major driver of the electronics industry in recent years is the growing demand for portable computing and communication devices. This demand has been fueled by the quality of life and business productivity improvements that have been provided by these devices. There has been a tremendous interest in laptop computers, personal digital assistants, mobile phones, and pagers. This is only the beginning, however, as there are more sophisticated devices on the horizon that will provide increasingly sophisticated capabilities and features. One important vision of the future of computing and communication has been proposed by the InfoPad project at the University of California at Berkeley [1, 2]. In this vision, mobile, wireless terminals provide users with ubiquitous and untethered access to multimedia content and computing services available from a high-bandwidth backbone network of computers.

A portable multimedia terminal must provide two fundamental capabilities: the ability to process multimedia information and the ability to communicate that information through various wired and/or wireless communications channels. Speech, audio, video,

and graphics are examples of the types of data that are processed by a typical multimedia terminal. The technology that provides the underlying algorithms to process these data types is Digital Signal Processing (DSP). Digital signal processing is also the technology that is applied to process the signals that are used to communicate information over a wired or wireless communication channel. Improvements in computational performance provided by advances in integrated circuit fabrication technology allow the use of more and more sophisticated signal processing techniques that allow greater functionality and performance and richer modes of communication in portable multimedia terminals. Speech recognition is a good example of the type of functionality that is currently not readily available, but it will be an important feature in the near future as the required processing power to provide it becomes economically viable. Another good example is a multi-standard, adaptive radio transceiver that can provide a number of different modes of communication, as required by the physical location of the user at a given time.

A major problem associated with increases in the processing power and the sophistication of signal processing algorithms is the increasing levels of power dissipation. A mobile terminal is typically powered by batteries, a limited source of energy. For a portable device to be useful, it must have a reasonable amount of run time before the batteries run out and need to be recharged. Another problem with high levels of power dissipation is the cost of packaging and cooling. Low-power integrated circuits can be placed in inexpensive and compact packages. High-power devices, on the other hand, require expensive and bulky packages and cooling mechanisms. High levels of power dissipation also mean high operating temperatures that adversely affect the reliability of an integrated circuit.

Power dissipation is not a new problem. In the middle of 1980s, designers faced the same problem. The solution then was to switch from NMOS technology, which suffered from static power dissipation, to CMOS technology [3, 4]. CMOS was far more

energy-efficient than NMOS and was the most effective and economically viable way to build more and more powerful microprocessors. In fact, CMOS was so energy efficient, that power became an afterthought once again. But in recent years, increasing levels of computing performance have made power an important and challenging problem once again [5], and this time, there is no magic technological solution to make it disappear.

To provide the required computing power and to increase the energy efficiency of signal processing circuits, designers have developed numerous design techniques that can be applied in custom, application-specific integrated circuits. While this approach has been successful in increasing energy efficiency, it suffers from the drawback that the resulting devices can only provide the limited functionality that they were designed to provide, i.e., they are not very flexible. In reality, however, a variety of multimedia data types and services, modes of communication, and associated standards are in use, and it is highly desirable to have devices that can deal with this variety. Therefore, it is highly desirable that flexible, programmable components be used to implement the processing functions required in a modern computing/communication device.

Programmability has many benefits, all of which are the results of the inherent flexibility of a programmable design. With a programmable device, one can use the same pre-fabricated component to perform different tasks. One does not have to go through the lengthy and costly cycle of designing a new integrated circuit that performs a new task. It is far easier, faster, and less expensive to program a programmable processor to perform a new task than it is to design a new integrated circuit.

Another advantage of a programmable implementation of an algorithm is that one can tune the parameters of a system by simply changing the parameters of a program. One can tune a design in its actual operating environment and get quick feedback as to how well design modifications work. Another advantage is that a system designed with pro-

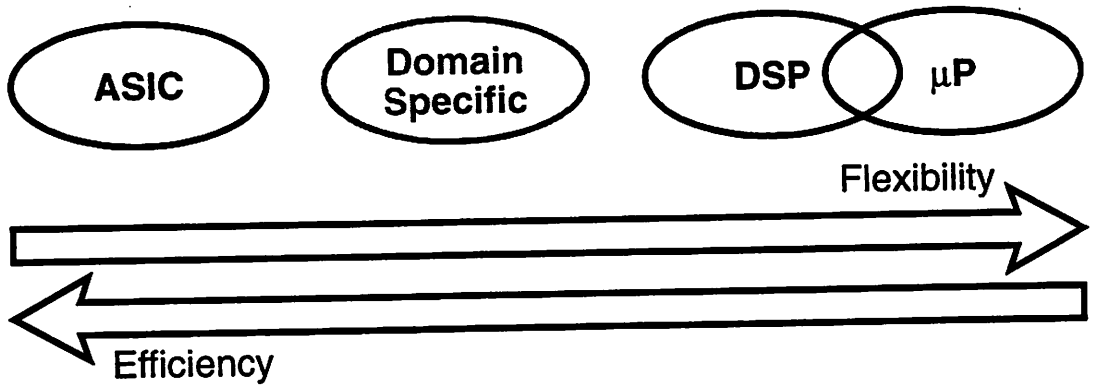


Figure 1.1: The Trade-off between Flexibility and Efficiency

grammable components can be upgraded during its lifetime to improve its functionality and to provide the ability to support new standards.

The difficulty in achieving high levels of energy efficiency (and performance) in a programmable processor stems from a fundamental trade-off that exists between flexibility and efficiency. This trade-off is illustrated in Figure 1.1. Programmability requires general-purpose computation, storage, and communication resources that can be used to implement all kinds of different algorithms. Efficiency, on the other hand, dictates the use of custom, dedicated hardware structures that can exploit the specific properties of a given algorithm to maximize efficiency. In a custom solution, no computational resource is larger or more complicated than it needs to be. As a result, all circuit modules, i.e., arithmetic and logic units, memories, and communication channels, are smaller, faster, and consume less energy.

Programmable processors, particularly general-purpose microprocessors and digital signal processors, have the virtue of being completely flexible. They can be programmed to implement any algorithm, but they incur the significant energy and performance overhead of fetching, decoding, and executing sequences of instructions on complex, general-purpose hardware structures.

1.1 Goals and Contributions

The central problem addressed in this work is how to design a digital signal processor that is not only highly energy efficient, but it is also programmable and can be used to implement a variety of different, but similar, algorithms. The flexibility of general-purpose processors is highly desirable for handling complex, control-oriented computing tasks such as operating systems, word processors, and spreadsheets. Signal processing algorithms, on the other hand, have intrinsic properties that provide an opportunity for creating more efficient implementations that do not require the full flexibility of a general-purpose device. Signal processing algorithms typically exhibit high levels of concurrency and are dominated by a few regular, repetitive kernels of computation that account for a large fraction of execution time and energy.

The approach taken in this work was to explore ways of trading off flexibility for increased efficiency. This approach was based on the observation that for a given class, or *domain*, of signal processing algorithms, e.g., speech coding using Code-Excited Linear Prediction (CELP) or video compression/decompression using the Discrete Cosine Transform (DCT), the underlying computational kernels that are responsible for a large fraction of execution time and energy are very similar. What varies in different algorithms and different industry standards are the parameters of the algorithms and the high-level control flow of the algorithms. By executing these underlying dominant kernels on dedicated, optimized processing elements that can execute those kernels with a minimum of energy overhead, significant energy savings can potentially be gained. This means that the processors being designed with this approach are *domain-specific* and are optimized for a given domain of algorithms. Flexibility is thus traded off, as illustrated in Figure 1.1, allowing a designer to achieve high levels of energy efficiency, approaching that of a custom, application-specific design, while maintaining the flexibility needed to handle a variety of different algorithms within the domain of interest.

The main contribution of this work is an architecture template, named Pleiades, that can be used to implement domain-specific, programmable processors for digital signal processing algorithms. Pleiades relies on a heterogeneous network of processing elements, optimized for a given domain of algorithms, that can be reconfigured at run time to perform different computational tasks. Associated with this architecture template is a design methodology. Defining this methodology was another contribution of this work. To explore and prove the effectiveness of this approach, a domain-specific processor for CELP-based speech coding algorithms, named Maia, was designed and analyzed. A prototype integrated circuit, named P1, with all the elements of the Pleiades architecture template was designed and fabricated to evaluate the merits of the Pleiades approach.

1.2 Thesis Overview

The body of knowledge that forms the background of this work will be presented in the next three chapters. Chapter 2 provides an overview of low-power design techniques for digital CMOS circuits. We will discuss how power is dissipated in CMOS circuits and how it can be minimized. The main objective of this chapter is to establish a set of architectural design principles that must be followed in an energy-efficient design.

Chapter 3 describes the properties of digital signal processing algorithms that can be exploited to design energy-efficient, domain-specific processors. A general overview of CELP-based speech coding algorithms and a detailed analysis of the VSELP (Vector-Sum Excited Linear Prediction) speech coding algorithm will be presented.

Chapter 4 presents a comprehensive review of the different approaches that have been explored in the past for designing programmable processors for digital signal processing applications. The strengths and weaknesses of these different architectures will be discussed. This chapter concludes with a set of architectural features that must be present in an energy-efficient programmable signal processor. These features, along with the

energy-efficient design principles presented in Chapter 2, form the basis for the design choices made in the Pleiades architecture template.

Chapter 5 presents the architecture template proposed in this research. We will first present the programming model that provides the skeleton of this architecture template, and we will sketch the associated design methodology. Next, the architectural design choices that were made will be presented and analyzed. We will show how signal processing kernels can be mapped onto the Pleiades architecture template. Architectural design of Maia, a domain-specific processor for speech coding applications, will be presented.

Chapter 6 presents the design of the P1 prototype which was designed and fabricated to evaluate the merits of the architectural principles presented in this thesis. We will show how different components of the Pleiades architecture template can be assembled into a practical design. Measured power and performance numbers will be presented and discussed.

The Pleiades approach is evaluated in Chapter 7. Benchmark results comparing the Pleiades architecture to other programmable architectures will be presented and discussed.

The last chapter concludes this dissertation with a summary of the presented work and proposals for future research.

CHAPTER 2

Principles of Low-Power Design

Programmable signal processors are typically implemented as digital integrated circuits using CMOS technology. In this chapter we will review the fundamentals of low-power digital CMOS design. We will start with a discussion of how energy is consumed in digital CMOS circuits. We will then discuss how energy consumption of digital CMOS circuits can be minimized. Architectural techniques for reducing power dissipation will be presented. We will end this chapter with a set of architectural design principles for energy-efficient programmable architectures.

2.1 Energy and Power

Energy and power are related. Power is the time rate of consumption of energy ($P = \dot{E}$). Electrical energy is consumed by a circuit to perform a given task, i.e., a computation, and is dissipated as heat and electromagnetic radiation. A circuit can be rated by the amount of energy that it consumes to perform a given task, or it can be rated by its power dissipation. Both of these ratings are useful in their own different ways. If we are concerned about battery life, then energy is the more appropriate metric to consider. A bat-

tery stores a finite amount of energy, and a finite amount of work can be done with that energy. What matters is to do as much work as possible; therefore, as little energy as possible must be consumed to perform a given task. If the work is done quickly, then power dissipation will be high; if the work is done slowly, then power dissipation will be low. In either case, the same amount of work has been done. The speed at which a task is performed, however, usually determines if the work being done is actually useful. In real-time signal processing applications, for example, an incoming stream of data must be processed at a specified rate. In this context, rating a circuit by its power dissipation is equivalent to rating it by its energy consumption. Still, the real objective is to minimize the energy consumed to perform a given task.

If we are concerned with heat removal and reliability, then power is the more appropriate metric to consider, as the heat generated by a circuit and its operating temperature are directly related to its power dissipation, and the amount of work being done is inconsequential. While minimizing energy per task is not the real objective in this context, it is still an appropriate design objective, as it will reduce power dissipation.

2.2 Power Dissipation in CMOS Circuits

Before discussing techniques for minimizing power, we need to understand how energy is consumed in CMOS circuits. There are two main components of power dissipation in a CMOS circuit: dynamic power and static power.

2.2.1 Dynamic Power

The most significant component of power dissipation in CMOS circuits occurs during switching transients, when the circuits are actually processing information. Figure 2.1 shows the circuit diagram of a CMOS inverter. The parasitic capacitances driven by the inverter have been lumped into a load capacitance C at the output of the inverter. There

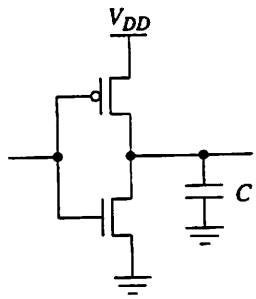


Figure 2.1: CMOS Inverter

are two mechanisms that result in dynamic power dissipation in this circuit. The first and most significant component of dynamic power is due to charging and discharging of the load capacitance. When the input of the inverter switches from high to low, the NMOS transistor is turned off, and the PMOS transistor is switched on, charging the load capacitance to V_{DD} and drawing CV_{DD} of charge from the power supply. As a result, CV_{DD}^{2} of energy is drawn from the power supply. Half of this energy is dissipated in the PMOS transistor, and the other half is stored in the load capacitance. When the input switches back to high, the PMOS transistor is turned off, the NMOS transistor is switched on, the load capacitance is discharged, and the energy that was stored on it is dissipated in the NMOS transistor. Thus, each switching event dissipates $\frac{1}{2}CV_{DD}^{2}$ of energy. If the operating frequency of the system within which this switching event is occurring is f, and the average number of switching events in this circuit during an execution cycle is α, then the power dissipation associated with this circuit is

$$P = \alpha f\left(\frac{1}{2}CV_{DD}^{2}\right) \tag{2.1}$$

The second component of dynamic power is caused by the non-zero transition time of the input of the inverter. In Figure 2.1, as the input is rising (or falling), there will be a

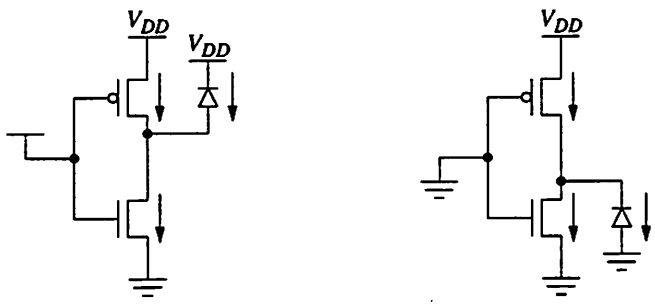


Figure 2.2: Leakage Currents in a CMOS Inverter

period of time during which both the PMOS and the NMOS transistors are on, and there is a direct path from V_{DD} to ground, allowing current to flow. Veendrick has done a detailed study of this mechanism, and his conclusion is that this component of power dissipation can be kept below 15% by maintaining equal transition times at the input and the output of a CMOS gate [6]. The contribution of this direct-path current to total power dissipation decreases as the supply voltage is reduced. In fact, if $V_{DD} < V_{TN} + |V_{TP}|$, where V_{TN} and V_{TP} are the threshold voltages of the NMOS and the PMOS transistors, respectively, then the direct-path current is virtually eliminated, as the two transistors cannot be on simultaneously. Thus, for low-power designs that are operated at low voltages, short-circuit power is not a major issue.

2.2.2 Static Power

In an ideal CMOS inverter, where the transistors are ideal switches, there is no static power dissipation because the PMOS and NMOS transistors are not simultaneously on in the steady state, and there is no DC path between the positive and negative terminals of the power supply. Real MOS transistors are not ideal switches, however, and in real CMOS circuits, there are two main mechanisms that result in static current flowing from the power supply [7]. Figure 2.2 illustrates these static currents in a CMOS inverter.

One type of static current is due to the junction leakage current of the reverse-biased diodes between the source and drain terminals and the substrate of a MOS transistor. This current is equal to the reverse saturation current of a PN junction diode, and is on the order of 1 to 5 pA per μm^2 of junction area at room temperature for a typical CMOS process [8]. For a minimum-size transistor in a 0.6-μm process, the total leakage current is on the order of 4 pA. The value of this current doubles for every 9 degree increase in temperature.

When the gate-to-source voltage, V_{GS}, of a MOS transistor is below its threshold voltage ($V_{GS} < V_T$), the transistor is considered off, and ideally, the drain current, I_D, of the transistor is zero. There is, however, a sub-threshold leakage current that flows through the device. This current decreases by an order of magnitude for every 60 to 90 mV drop in V_{GS}. Thus the drain current of an off device ($V_{GS} = 0$) is several orders of magnitude smaller than the operating current when the device is on. It should be noted that for reliability and power reasons, modern sub-micron processes operate at reduced supply voltages that dictate reduced threshold voltages (on the order of 400 mV). As a result, sub-threshold currents have become the dominant source of static leakage currents in modern sub-micron technologies. For example, for a 0.35-μm process, the sub-threshold current is on the order of 9 pA per μm of device width [9]. The value of the sub-threshold current also increases with temperature exponentially.

During normal operation, the power dissipation of a CMOS circuit due to the leakage currents is negligible, as these currents are orders of magnitude less than the operating currents when devices are switching. When a CMOS circuit is in stand-by, though, its power dissipation is determined by these leakage currents. Thus, if a circuit spends a large fraction of its operating time in stand-by, static power can become important.

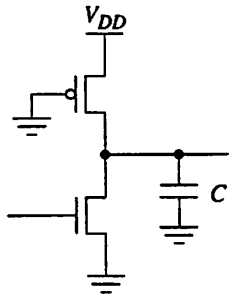


Figure 2.3: Pseudo-NMOS Inverter

There is another source of static power in CMOS circuits that occurs in ratioed circuit styles. Figure 2.3 shows the circuit diagram of a Pseudo-NMOS inverter. In this circuit, when the input is high, both transistors are on, and static current flows through the gate. In general, circuits of this sort must be avoided in energy-efficient designs, so that there are only leakage currents when there is no switching activity.

2.2.3 Summary

From the above discussion, we can see that the energy required to perform a given computation is determined by the switching energy consumed in charging and discharging of circuit nodes. As shown in Equation 2.1, this energy depends on three parameters: supply voltage, capacitance, and switching activity. In the following sections, we will study the effect of these parameters on energy consumption, and we will discuss energy-efficient design techniques and their effect on these parameters. Since we are primarily concerned with energy-efficient architectures in this thesis, the emphasis will be on architectural design techniques. It should be noted, however, that many other design techniques addressing other levels of the design process such as algorithm design, logic design, circuit design, and technology design have been proposed [10]. To minimize energy consumption all aspects of the design process must be energy conscious.

2.3 Reducing the Supply Voltage

Since power dissipation varies with the supply voltage in a quadratic manner, reducing the supply voltage is a very effective way of reducing power dissipation. For example, if the supply voltage is halved, then the power dissipation of an integrated circuit is reduced by a factor of four! Because of this quadratic relationship, reducing the supply voltage is *the* most powerful approach to reducing power dissipation.

Unfortunately, the supply voltage of a circuit cannot be reduced arbitrarily. As in most engineering problems, there is a trade-off at work that prevents us from arbitrarily reducing power dissipation by simply reducing the supply voltage. The problem is that the delay of CMOS circuits increases as the supply voltage is reduced. The drain current of a MOS transistor in saturation is

$$I_D = \frac{k'}{2}\frac{W}{L}(V_{DD} - V_T)^2 \tag{2.2}$$

where k' is the device transconductance parameter, W is the channel width, L is the channel length, and V_T is the threshold voltage. I_D decreases as V_{DD} approaches V_T. Thus, at lower voltages, the current level provided by the transistors to charge and discharge circuit nodes decreases, and circuit delays increase significantly. Figure 2.4, shows how the delay and energy of an inverter circuit vary with the supply voltage in the 0.5-µm CMOS process that was used in this research project. Performance degrades rapidly when the supply voltage is lowered beyond 1.2 V. Almost all designs have a minimum performance requirement, and in general, the supply voltage should be set at the minimum value that provides acceptable performance. A good metric for comparing the energy efficiency of different designs is the energy-delay product [11]. This metric captures the trade-off that a designer can make between performance and energy efficiency. The graph for the energy-

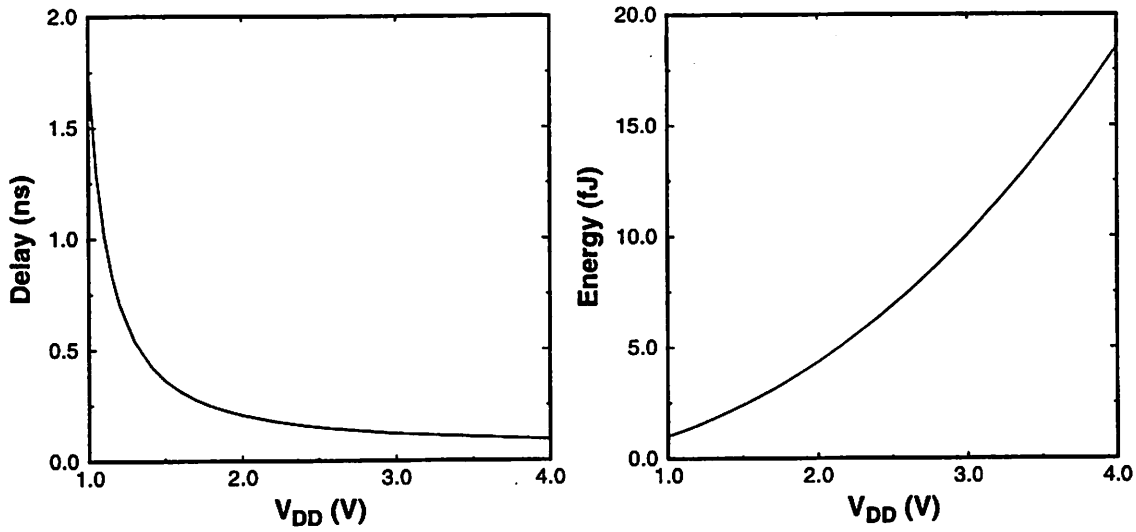


Figure 2.4: Dependence of Delay and Power on Supply Voltage

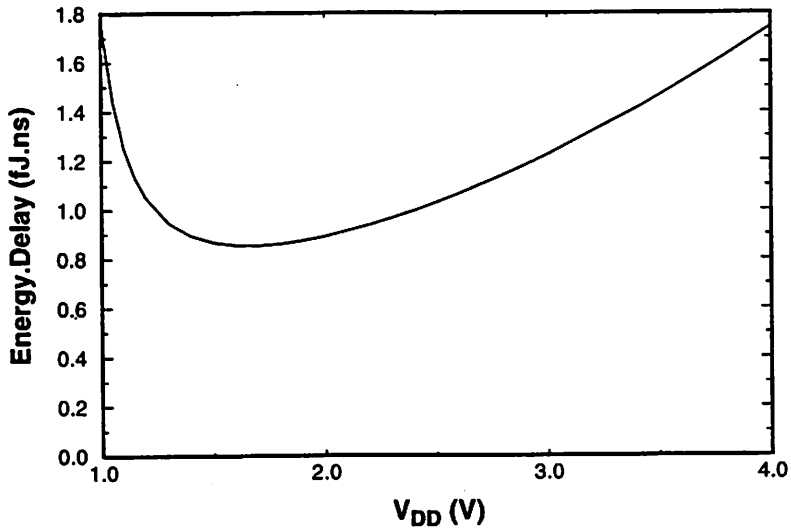


Figure 2.5: Energy-Delay Product vs. Supply Voltage

delay product of the inverter circuit mentioned above is shown in Figure 2.5. The nominal supply voltage for the circuits designed in this research project was 1.5 V. This design point is very close to the minimum of the energy-delay curve.

The golden rule in minimizing power dissipation is to design systems that can run at as low a supply voltage as possible that will satisfy the performance requirements. The choices made at all design levels, from algorithms and architectures to circuits and technologies should allow the reduction of the supply voltage as much as possible. This means that these design choices must be able to cope with and compensate for the speed loss associated with reducing the supply voltage. Some of these choices might result in more physical capacitance and/or more switching activity, but if they allow a reduction in the supply voltage, then the quadratic decrease in power may more than compensate for the increase due to the increased physical capacitance and switching activity.

One approach to further reduce power dissipation is to run each circuit at its own optimal supply voltage, which could be different from that of other circuits [12, 13]. This approach requires routing of multiple supply lines to different blocks of a chip, and it also requires level-shifter circuitry that will allow translation of signal levels between two blocks that run at different supply voltages. The overhead of these level-shifters and the complexity of the extra routing will limit how far this approach can be taken. Nevertheless, partitioning a chip into two or three voltage domains that have different performance requirements can be very effective in reducing power dissipation.

2.3.1 Concurrent Processing

Concurrent processing is a well-known architectural technique that can be used to increase the processing throughput of a design. This increase in throughput can be used to compensate for the speed loss associated with lowering the supply voltage. By applying concurrent processing techniques and thus increasing the throughput of a given design, we can lower the supply voltage and reduce power dissipation, while still meeting performance requirements [5, 8]. This approach can be used if the algorithm being executed by our design can be divided into separate tasks that can be executed concurrently. As we will

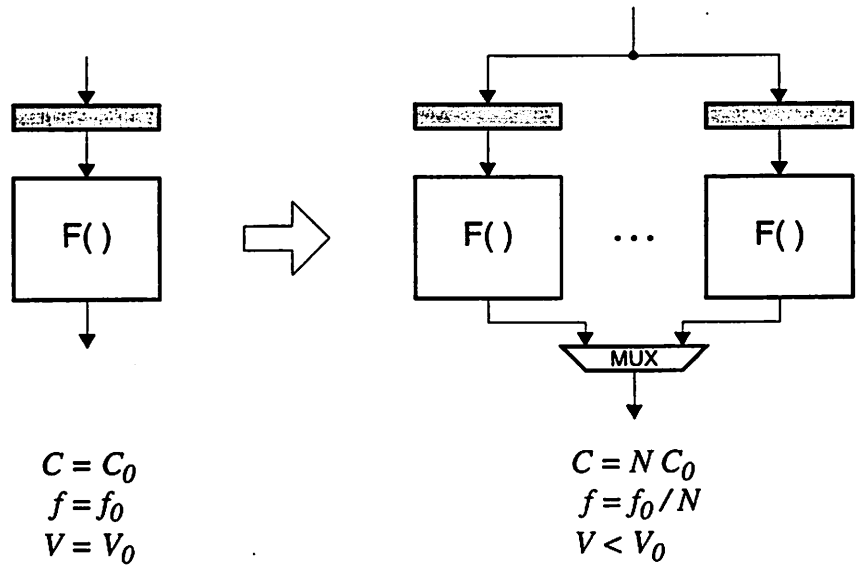


Figure 2.6: Parallel Processing

see in the next chapter, signal processing algorithms exhibit high levels of concurrency that can be exploited in this manner to reduce power dissipation.

There are two methods to realize concurrent processing: parallel processing and pipelining. In parallel processing, a functional unit is replicated N times. The input data stream is distributed to the functional units, and each functional unit operates on one token of input data in parallel with others. This is illustrated in Figure 2.6. In the parallel design, N tokens of input data are processed concurrently, and the throughput of the original design with a single functional unit has been increased by a factor of N. Capacitance has increased by a factor of N, but we can now lower the clock frequency by a factor of N. To meet the original performance requirement, each functional units can now operate N times slower than before, and we can lower the supply voltage and benefit from the quadratic drop in the power dissipation. The area of the design has increased by a factor of N, however, so in effect, we have engaged in an area vs. power trade-off. The area increase is one factor that limits how large N can be. Another factor that limits N is the capacitance and

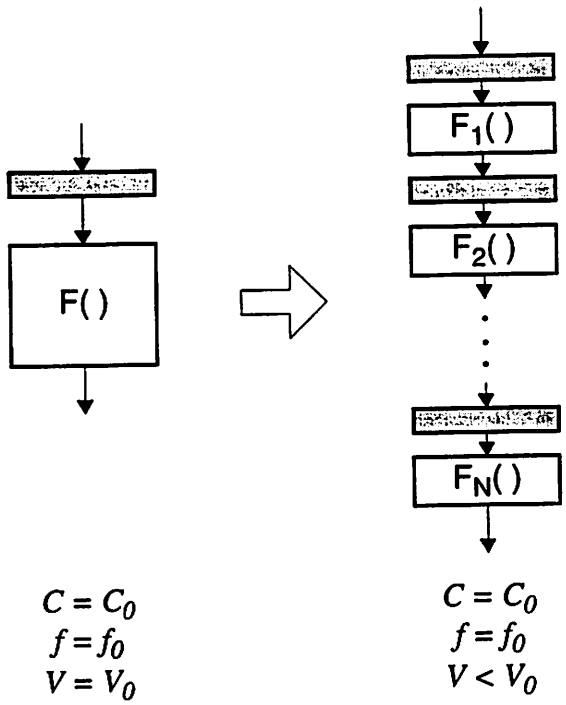


Figure 2.7: Pipelined Processing

delay overhead of distributing data tokens to and merging data tokens from the parallel functional unit. This overhead can become prohibitively large as N increases and limits how far this trade-off can be taken.

Pipelining is the other method to realize concurrent processing. In a pipelined design, each functional unit is divided into multiple stages. The pipeline stages are separated by registers, and each stage can operate on a different data token concurrently. This is illustrated in Figure 2.7. The cycle time of the pipelined design is equal to the logic delay of the slowest stage. In an optimal N-stage pipeline, the delay of each stage is $1/N$ of the original delay, and processing throughput increases by N. Ignoring the pipeline registers for the moment, we can see that the capacitance has not changed, and to meet the original performance requirement, the clock frequency does not change, either, but since

the logic depth between registers has decreased, we can lower the supply voltage and benefit from the quadratic reduction in power dissipation. In practice, the delay and capacitance of the pipeline registers limit how far this technique can be taken. Another limitation is that the pipelined design has an additional $N-1$ cycles of latency in producing a result compared to the original design, and in some applications this may not be acceptable.

2.3.2 Dynamic Scaling of the Supply Voltage

In some applications, the performance requirements of a system may vary in time. In these applications, it will make sense to dynamically vary the supply voltage, so that it is always at the lowest possible value that provides sufficient throughput. If a system spends most of its time in the low-throughput mode, then the potential savings of adaptively scaling the supply voltage can be significant. A good example of an application where this technique can be very effective is in the error-corrector circuitry of a digital compact cassette (DCC) audio player [14]. In normal operation, when error rates are low, the required throughput is also low, so the system can run at a low supply voltage. When there is a burst of errors, the error-correction circuits must perform a large number of additional computations to correct those errors in real-time. When that happens, the supply voltage can be automatically increased to provide sufficient performance to correct the errors. Another important application where dynamic scaling of the supply voltage can be very effective is in the microprocessor circuits of battery-operated portable computers [15]. The supply voltage can be dynamically adjusted by the operating system based on the amount of work being executed by the computer.

2.3.3 Reduced-Swing Interconnect

Equation 2.1 was derived under the assumption that the voltage swing on a circuit node switching from high to low (or from low to high) is equal to V_{DD}. This is indeed the

case in CMOS circuits where $V = 0$ represents the low logic level, and $V = V_{DD}$ represents the high logic level. If the voltage swing is V_{sw}, instead, then the switching energy is

$$E = \frac{1}{2} C V_{sw} V_{DD} \tag{2.3}$$

Thus, by reducing the voltage swing we could reduce switching energy linearly. This requires the use of special driver and receiver circuits that can produce and sense reduced swings [16]. This technique can be used only when the energy savings are far more than the overhead incurred by the driver and receiver circuits. For circuit nodes that are heavily loaded, the energy (and propagation delay time) saved by using reduced swings can be significant. This technique can be especially valuable in programmable processors where numerous buses are used to carry information between computational and storage blocks.

2.4 Reducing Capacitance

Since switching energy is proportional to the capacitance of a switching circuit node, minimizing capacitance is an important goal for reducing power dissipation. Node capacitances are due to the parasitic capacitances of the transistors and the wires. Transistor capacitances are due to the gate capacitance and the diffusion capacitances of the source and drain areas. Gate capacitance is proportional to the area of the gate, and the area of the gate is equal to the product of the transistor width and the channel length. Since in digital circuits, channel lengths are typically at the minimum allowed by the fabrication technology, then gate capacitance is proportional to the width of the transistors. Diffusion capacitance has a *bottom* and a *side-wall* component which are proportional to the area and perimeter of the diffusion areas, respectively. It follows that diffusion capacitance is also proportional to transistor width. Wire capacitance is proportional to the length of a wire. It follows that for low-power design we must try to minimize the size of the transis-

tors and the length of the wires. There is, however, a trade-off at work here that should be kept in mind for a successful design. Reducing the size of the transistors slows down the circuits. In some situations, it is advantageous to use larger devices, and hence increase capacitance, but since the circuits are faster, we can operate them at a lower supply voltage and benefit from the resulting quadratic drop in power dissipation. In this scenario, even though we have increased capacitance, we have reduced the overall power because we have managed to run the circuits at a lower supply voltage. This approach should be taken with critical circuit paths that determine the throughput of a design. Circuits on the non-critical paths should use the smallest possible devices.

2.4.1 Application-Specific Processing

One approach to minimizing capacitance is to use circuit blocks that are custom-made to perform the specific computational tasks required by a given application. In this approach, the use of more versatile and general-purpose circuit blocks is to be avoided. This approach can significantly reduce the capacitance associated with an operation because an application-specific circuit block is no larger and no more complicated than the bare minimum required to execute the required operation. General-purpose circuit blocks are necessarily larger and more complex because they are designed so that they can execute several different operations. They also have to be large enough to handle the largest data size encountered in a given application. For example, it is far more efficient to add two 8-bit operands on an 8-bit adder than it is on a 16-bit general-purpose arithmetic/logic unit. While the 16-bit ALU is versatile and can execute other useful operations and can also handle the longer word lengths that may be present in the application at hand, it is very inefficient for adding two 8-bit numbers. If in a given application, most of the operations executed by this ALU are 8-bit additions, then a great deal of energy is wasted. As we will see in Chapter 4, one of the reasons why general-purpose processors are so much less energy efficient than application-specific designs is that they waste a great deal of

energy in large, centralized computational resources that are designed to be completely general-purpose.

2.4.2 Exploiting Locality of Reference

Driving global signals across a chip and accessing large, centralized memories and functional units are power-consuming tasks that must be avoided in an energy-efficient design. This can be accomplished by partitioning a design such that the *locality of reference* present in a given algorithm is preserved.

An algorithm consists of a sequence of computational steps. Each step of an algorithm uses one or more operands produced in previous steps and produces new operands that are used by the following steps. Locality of reference is a natural property exhibited by many algorithms and arises from the fact that most computational steps typically interact and communicate with only a few previous and subsequent steps. Communication patterns in the data flow graphs of these algorithms are localized, and it is very rare that a computational step communicates globally with many other steps. By partitioning a system properly, this locality can be exploited to minimize the amount of power-hungry global interactions. This can be achieved by a *distributed processing* approach in which, instead of using a single, centralized general-purpose processor, the computations required by a given algorithm are distributed across a set of smaller local processors. This approach can significantly reduce the power associated with data transfers. An additional benefit of this approach is that the local processors can be optimized for a particular section of the algorithm and can thus be far more energy efficient than a single, centralized general-purpose processor.

An additional benefit of distributed processing is that the energy of memory accesses can be significantly reduced. This is particularly important because memory accesses can be responsible for a significant fraction of total power dissipation [17, 18].

The energy of a memory access is proportional to the number of words stored in that memory. A distributed array of small, local memories can, therefore, be far more energy-efficient than a single large, shared memory.

Another aspect of distributed processing is the use of distributed controllers. In a centralized control approach, a single finite-state machine generates all control signals for all processors and memories. The energy overhead of distributing these signals across the chip can be significant. In a distributed control approach, only a small amount of global control information is distributed to local controllers, which then generate all of the control signals required locally.

2.5 Reducing Switching Activity

Since switching events are the cause of energy consumption, in an energy-efficient design the number of switching events must be minimized. In other words, any given computation must be performed with a minimum number of switching events. There are a number of ways that excess switching can be avoided.

Ideally, during every execution cycle, since each logic gate generates one result, there should be at most one switching event at the output of each gate if the logic output of the current processing cycle is different from that of the previous cycle. In combinational CMOS gates, however, the output of a gate can switch multiple times before it settles to its final value. This effect is called *glitching*, and it is caused at circuit nodes whose logic function is a function of a number of inputs with different path delays leading to the gate driving the circuit node in question. As the results from these different paths arrive at the gate one at a time, the gate evaluates several times until all inputs have arrived, and the gate then produces it final output. This mechanism can actually waste quite a bit of energy, especially in structures where there are many different paths leading to the outputs. One good example is a carry-ripple adder. As the carry signal ripples through the adder, the

outputs can glitch many times, as several intermediate values are evaluated. A designer should carefully analyze a design and pick logic structures that have more balanced paths in order to minimize glitching. One technique is to insert extra delays to create more balance in the logic structure [19].

How data is represented and encoded can have an important effect on the amount of switching activity, as well. The reduced switching activity of a given representation can more than compensate for a possible increase in circuit complexity and capacitance. For example, the sign-magnitude representation can result in less switching activity than the familiar two's-complement representation [20]. In the two's-complement representation, when the sign of a value changes, several of the most significant bits can change. In sign-magnitude representation, however, only the most significant bit changes. For example, in the transition from 0 to -1, all of the bits will change when numbers are represented in two's-complement format ($00000000 \Rightarrow 11111111$), whereas in sign-magnitude representation, only two bits change ($00000000 \Rightarrow 10000001$). Arithmetic circuits in sign-magnitude are, however, more complex, so it may not always be beneficial to use the sign-magnitude representation. But if the data in question is being transmitted through a heavily loaded bus, then using the sign-magnitude representation can save energy.

2.5.1 Avoiding Switching Activity in Unused Modules

An important approach in low-power design is to avoid any kind of unnecessary switching activity. This approach corresponds to a design philosophy in which all circuit activities occur strictly in a *demand-driven* fashion. This means that no circuit node should ever switch unless there is an actual demand for it. This seemingly simple objective can, however, be quite difficult to achieve. A number of different techniques have been developed to minimize excess switching activity.

In a conventional synchronous digital system, a global clock signal synchronizes the transfer of data to the storage elements, i.e., registers and latches. The clock signal is distributed across the entire chip and triggers all registers and latches. The clock signal is, thus, a heavily loaded signal and can consume a great deal of power. Even if there is no new input data to be processed by the system, the clock signal is still switching and the storage elements are being clocked. This can be a significant waste of power if a system spends only a fraction of the time performing useful work. An important objective in low-power design, then, is to prevent this unnecessary switching activity.

A useful technique to reduce unnecessary switching activity is to use *gated clocks*. In this approach, additional control logic is used to monitor the activity of different modules in a chip. This control logic determines if a given module is needed to do useful work and produces control signals that can gate off the clock signal going to that module when it is not needed. Thus, no energy is wasted in an idle module. This approach can be quite effective in reducing unnecessary switching activity. This approach can be applied down to the level of individual storage elements [21], but the overhead of the required control logic must be carefully taken into account, as it may not always be beneficial to add clock control circuitry for every single register and latch. Since gated clocks introduce additional logic in the clock signal path, they can complicate the distribution of the clock signal across the chip. Extra design effort is required to minimize clock skew between different clock domains.

With gated clocks, while idle modules are deactivated and waste no power, there is still the power consumption of the free-running global clock signal which can be significant. A common approach to avoid unnecessary power consumption by the clock signal is to monitor the activity of the system and to deactivate the entire chip after a specified period of inactivity. During this *sleep mode*, the inputs of the system must be monitored to

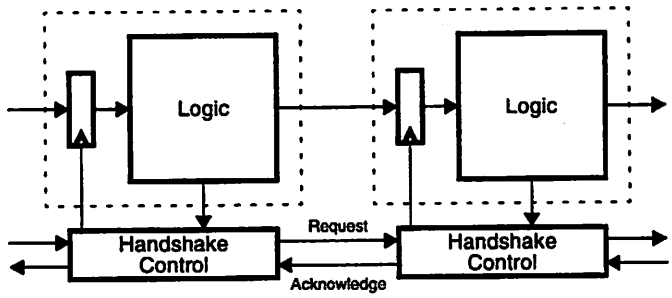


Figure 2.8: Asynchronous Processing with Handshake Control

determine when to reactivate the system. This approach can be quite effective in systems that spend most of their time in the idle mode, e.g., cellular phones, but its application is highly dependent on the nature of the application in question. An important design issue is the latency associated with switching into and out of sleep modes that must be carefully considered.

A radically different approach to minimizing unnecessary switching is to use asynchronous circuits. In asynchronous, or self-timed, systems there is no global clock signal that is distributed across the chip. The clock signals of the storage elements are, instead, generated locally under the control of handshaking circuits that coordinate data transfers between different modules (see Figure 2.8). Arrival of new data at the inputs of a given module is accompanied by a request signal that activates the circuitry in that module. An important benefit of asynchronous circuits is that they have a built-in, automatic power-down capability. Arrival of new data triggers new activity, and when there is no new data to be processed, there is no switching activity. An additional benefit of asynchronous systems is that the power overhead of distributing a global clock signal is avoided, as there is no global clock signal. These properties make asynchronous circuits attractive for low-power systems. We will discuss asynchronous circuits in more detail in Chapter 5, where a locally-synchronous/globally-asynchronous approach is presented.

2.5.2 Exploiting Temporal Correlations

The data streams processed by a signal processing system correspond to physical signals such as voice or video signals. These signals represent continuous functions, and they typically vary slowly compared to the rate at which they are sampled. As a result, each sample of such a signal is highly correlated with its neighboring samples. In other words, such a signal exhibits a great deal of *temporal correlation*. Temporal correlations are not limited to data streams representing physical signals, though. They can exist in other kinds of data streams such as address sequences for accessing regular data structures such as vectors and matrices. The program counter of a microprocessor, for example, produces instruction address streams that are highly correlated.

The amount of switching activity caused by a correlated data stream can be significantly less than that of a random sequence of uncorrelated samples [8, 22]. When this data stream propagates through various processing modules of a system, it results in less switching activity than an uncorrelated, random data stream. This property can be exploited to reduce switching activity by avoiding architectures that can destroy these temporal correlations. Temporal correlations are destroyed when hardware resources are time-shared to process multiple data streams in a multiplexed fashion. A time-multiplexed processing element alternates between multiple input data streams on a cycle-by-cycle basis. Therefore, each sample processed by this element belongs to a data stream that is different from that of the previous and the following samples. The net result is that the actual data stream processed by this element has no temporal correlations, and as a result, switching activity can increase significantly. A low-power architecture should, therefore, try to exploit temporal correlations in data streams by avoiding time-sharing of hardware resources. An additional drawback of time-sharing of hardware resources is that it limits the extent of supply voltage reduction because it requires that processing elements be clocked faster than they would be if they did not have to process multiple streams of data.

2.6 Summary

We end this chapter with a list of design principles that must be followed in designing low-power systems:

- To minimize the supply voltage concurrent architectures that can support parallel and pipelined processing are required. This is, by far, the most effective approach to minimize the supply voltage.
- Partitioning a system into a small number of voltage domains is an effective technique to minimize overall power while providing higher performance in processing elements that are timing-critical. Special circuits that can translate signal levels between different voltage domains must be used.
- Dynamic scaling of the supply voltage must be supported. This technique can be particularly effective in applications where periods of high-throughput processing come in bursts.
- The voltage swing on the communication links between processing elements must be minimized. This requires the use of special driver and receiver circuits that can operate with reduced voltage swings.
- To minimize the capacitance associated with basic computational steps, application-specific processing modules that have been optimized for the common operations of a given algorithm must be used. Large, general-purpose processing elements and memories must be avoided.
- Locality of reference must be exploited to minimize capacitance. Large, centralized hardware structures must be abandoned in favor of structures that support distributed processing. Increased concurrency is a beneficial side-effect of this approach.
- Unnecessary switching activity must be avoided. This can be achieved by system-level power-down modes and gated clocks. Asynchronous processing can be par-

ticularly effective to minimize switching activity because it exhibits built-in automatic power-down of unused modules.

- Time-sharing of hardware resources destroys the temporal correlations present in data streams and must be avoided. This is particularly important in signal processing applications. It should be noted that this approach is consistent with the goal of supporting concurrent processing.

To approach the energy efficiency of a custom, application-specific integrated circuit, it is imperative that *all* of the principles listed above be applied aggressively in designing a programmable architecture.

CHAPTER 3

Properties of Digital Signal Processing Algorithms

In this chapter we will take a look at some important properties of digital signal processing algorithms that must be considered when designing programmable architectures for these algorithms. This will be done by studying the characteristics of some example algorithms. We will then present an overview of speech coding algorithms that are based on Code-Excited Linear Prediction (CELP), and we will study and analyze the Vector-Sum Excited Linear Prediction (VSELP) algorithm in detail. This chapter will conclude with a list of architectural requirements that must be satisfied in designing efficient programmable architectures for signal processing algorithms.

3.1 Computational Performance Requirements

DSP applications are real-time in nature and involve processing of input signals that arrive at a specified sample rate. For example, audio signals in digital compact disc applications are sampled at 44.1 kHz [23]. Sampling rates for video signals are typically in the range of 10's of MHz. This means that any given implementation of a signal processing algorithm must have sufficient computational performance to process the incoming

data streams at the specified rate (and no faster). As a result, DSP applications tend to have large performance requirements ranging from 10's of MOPS (Million Operations Per Second) for speech and audio applications to 10's of GOPS (Giga Operations Per second) for video applications.

Another factor contributing to the required computational performance level for a given application is the complexity of the processing that is performed. A good measure of the complexity of a signal processing algorithms is the number of operations per sample of the input signal. Speech coding applications tend to exhibit a great deal of complexity that can be in the range of 100's of operation per sample of speech. Video applications on the other hand tend to exhibit less complexity that is typically on the order of 5 to 10 operations per sample.

3.2 Concurrency

One of the key properties of signal processing algorithms that has a major impact on architecture design is the abundance of concurrency in signal processing algorithms. Signal processing algorithms exhibit high levels of *spatial* and *temporal* concurrency that can be exploited by parallel and pipelined processing, respectively. This is quite fortunate, as the high levels of concurrency exhibited by DSP algorithms can be exploited to meet the computational performance levels demanded by these algorithms. As we saw in Chapter 2, exploiting concurrency is an important means of lowering the supply voltage and reducing the power dissipation of an architecture. An efficient programmable architecture for signal processing algorithms must, therefore, be able to exploit the concurrency present in these algorithms. One aspect of exploiting this concurrency is having multiple processing units that can perform multiple computations concurrently. Another aspect is to provide these processing units with the increased memory bandwidth that is a result of processing multiple operands concurrently. Not only the structure of the data store must

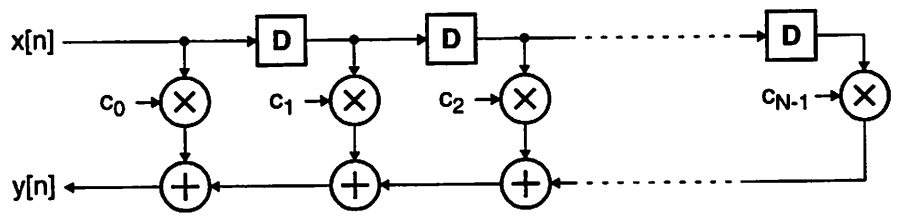


Figure 3.1: Finite Impulse Response Filter

allow concurrent accesses, but the interconnection network between the data store and the processing units must allow concurrent transport of all the required operands during a given execution cycle. Additionally, the interconnect between the functional units and the memories must be flexible enough to support the communication patterns that typically arise in DSP algorithms. In the next two subsections, we will illustrate the abundance of concurrency in DSP algorithms by considering two important examples that are very common in many DSP applications. While we are considering only two examples, it should be pointed out that the characteristics illustrated by these examples are common across a vast majority of, if not all, DSP algorithms.

3.2.1 The Finite Impulse Response Filter

The finite impulse response (FIR) filter is one of the most common algorithms in signal processing. The computation associated with an N-tap FIR filter is described by the following difference equation:

$$y[n] = \sum_{i=0}^{N-1} c_i \cdot x[n-i] = c_0 \cdot x[n] + c_1 \cdot x[n-1] + \ldots + c_{N-1} \cdot x[n-N+1] \quad (3.1)$$

The block diagram associated with this computation is shown in Figure 3.1. The spatial concurrency exhibited by this algorithm can be readily seen in this diagram. All N multiplications can be performed in parallel in $O(1)$ time. The N additions can also be

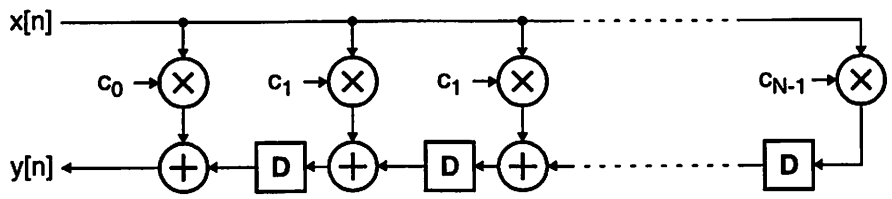


Figure 3.2: Retimed FIR Filter

done in parallel using a tree structure in $O(\log N)$ time. This algorithm also exhibits temporal concurrency which can be exploited by retiming [24] the algorithm as shown in Figure 3.2. In this retimed version, each multiplication and the addition that follows it form a pipeline stage that executes concurrently with other pipeline stages, and as a result, the FIR computation is executed in $O(1)$ time. As we can see, the FIR filter exhibits a high degree of spatial and temporal concurrency, and the throughput of an N-tap FIR filter can be increased by a factor of N if this inherent concurrency can be fully exploited by a given implementation.

3.2.2 The Fast Fourier Transform

The Fast Fourier Transform (FFT) is an efficient, divide-and-conquer algorithm for calculating the Discrete Fourier Transform (DFT) of a discrete-time sequence [25]. The DFT of a finite-duration sequence $x[n]$ of length N $(0 \le n \le N-1)$ is a finite sequence $X[k]$ of length N $(0 \le k \le N-1)$ defined as

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn} \tag{3.2}$$

$$\text{where } W_N = e^{-j(2\pi/N)} \tag{3.3}$$

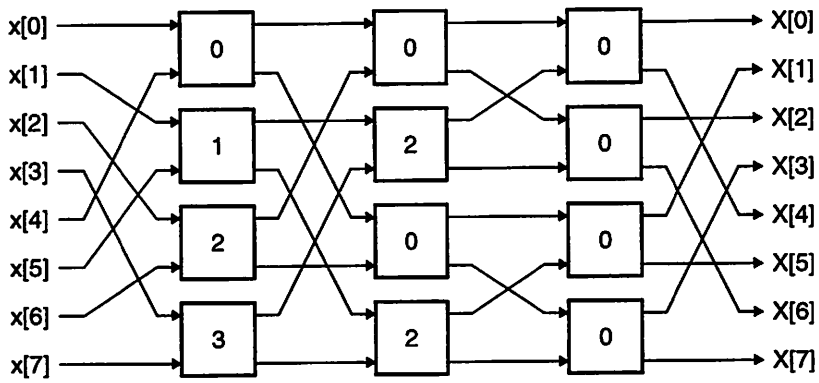


Figure 3.3: 8-Point, Radix-2, Decimation-in-Frequency FFT

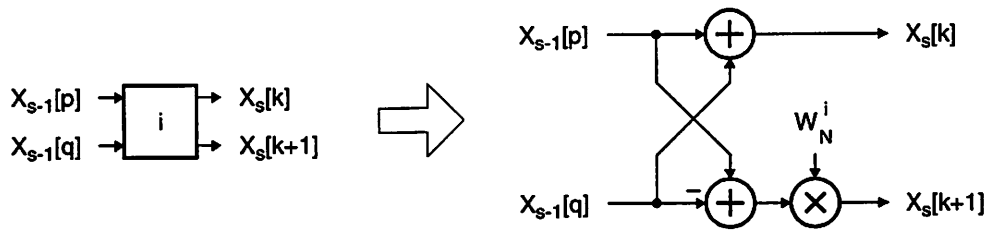


Figure 3.4: Radix-2 FFT Butterfly Computation

The time complexity of a straightforward calculation of the DFT as defined above is in $O(N^2)$, whereas the time complexity of the FFT algorithm is in $O(N \log N)$. The most popular version of the FFT algorithm is the radix-2 algorithm, for which N is a power of 2. The block diagram of an 8-point, radix-2, decimation-in-frequency (as opposed to decimation-in-time) FFT is shown in Figure 3.3. The computation performed by the blocks in this diagram is known as the *FFT Butterfly* and is illustrated in Figure 3.4. In general, an N-point, radix-2 FFT consists of $\log_2 N$ processing stages, each of which involves $N/2$ butterfly computations. Each stage generates a vector $X_s[k]$ of length N by processing the vector $X_{s-1}[k]$ generated by the preceding stage. The output of the last

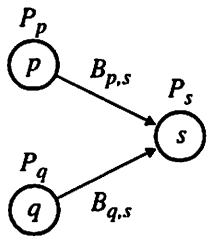
stage is the DFT of the input sequence. Since the elements of these vectors are complex numbers, the butterfly computation consists of four real multiplications and four real additions. As we can see in Figure 3.3, all butterfly computations of a given stage of the FFT algorithm can be performed in parallel. In addition, the algorithm can be pipelined by inserting registers between the stages. The butterfly computation also exhibits fine-grain concurrency that can be exploited by parallel and pipelined processing. Thus, the FFT algorithm exhibits a great deal of concurrency that can be exploited to create an efficient design.

3.3 Dominant Kernels

An important property of signal processing algorithms is that their execution time (and energy) is dominated by regular, repetitive *kernels* of computation. These kernels are the calculations that are performed in the inner loops of a program implementing a given DSP algorithm. We saw two examples of these dominant computational kernels in the previous section. The dominant kernel in the FIR filter is the tap calculation which is a multiply-add operation. The multiply-add (also known as multiply-accumulate or MAC) operation is in fact one of the most common kernels in signal processing and appears in a wide variety of algorithms. The dominant kernel in the FFT algorithm is the butterfly calculation. The FFT algorithm is in essence nothing but a collection of butterfly operations. The mean-squared error (MSE) calculation is another example of a dominant kernel which is commonly used to represent the magnitude of the difference between two vector quantities:

$$e_{MS} = |\mathbf{A} - \mathbf{B}|^2 = \sum_{i=1}^{N-1} (A_i - B_i)^2 \tag{3.4}$$

In vector quantization algorithms [28], where the objective is to select one of a set of vectors that is the closest to a given input vector (representing an image block or a frame of



$$P_s(n) = \min[P_p(n-1) + B_{p,s}(n-1)\,,\, P_q(n-1) + B_{q,s}(n-1)]$$

Figure 3.5: Radix-2 Viterbi Add-Compare-Select Calculation

speech, for example), the mean-squared error calculation is a significant fraction of the total execution time. Another example of a dominant kernel is the add-compare-select (ACS) calculation of the Viterbi algorithm [26, 27], which is widely used in digital communication and magnetic storage systems. The ACS calculation is illustrated in Figure 3.5. The objective here is to decide which state transition to state s ($p \rightarrow s$ or $q \rightarrow s$) will minimize the path metric for the path leading to state s (P_s).

Executing the dominant kernels of a given signal processing algorithm on efficient hardware structures that can execute these kernels with a minimum of energy overhead can save significant amounts of energy, as most of the execution energy is consumed by the dominant kernels. This is one of the key factors that makes a custom, application-specific implementation of a DSP algorithm highly energy-efficient. A typical programmable processor, however, does not exploit the opportunities presented by the dominant kernels of a DSP algorithm and incurs a great deal of energy overhead in executing those kernels.

3.4 Data Structures and Access Patterns

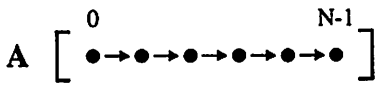
In addition to kernel calculations that process input data and generate output data, i.e., *data* calculations, the inner loop implementing a given kernel has to perform addi-

tional calculations which are considered *address* calculations. These calculations are performed to determine the memory addresses of the required input and output data tokens. The mathematical formalism used in describing signal processing algorithms deals with vector and matrix quantities, and as a result, in addition to scalar quantities that are usually stored in local registers, the data structures manipulated by signal processing algorithms are usually one- and two- (and sometimes higher) dimensional arrays that are stored in memory. The data calculations of a kernel are therefore accompanied by address calculations that are used to determine the memory addresses of the data elements required by the data calculations.

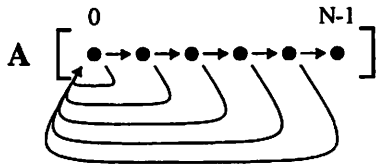
How address calculations are handled is an important architectural issue. For a general-purpose processor, address calculations are no different than data calculations. A general-purpose processor can handle all calculations equally well (or equally badly!) with its general-purpose datapath under program control, but this is not necessarily the most optimal approach. Signal processing algorithms tend to access array variables in a very structured manner. As an inner loop executes, array variables are scanned in orderly patterns. Some examples of access patterns are shown in Figure 3.6. While in principle there are many different ways of scanning the elements of an array, and flexible address generators are needed in a programmable architecture, the access patterns that are commonly encountered in practice can be implemented efficiently by simple arithmetic and logic operations on address pointers.

3.5 Case Study: Speech Coding by Code-Excited Linear Prediction

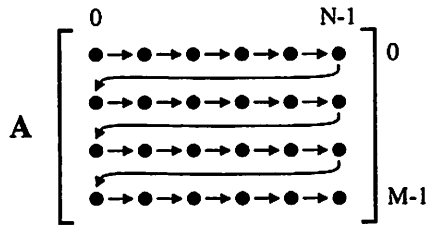
The problem addressed by *speech coding* is that of reducing the amount of information required to describe speech signals [29]. A central issue in voice communication applications, e.g. telephony, is the amount of bandwidth required to represent speech signals adequately. It is experimentally known that the power spectrum of the human speech



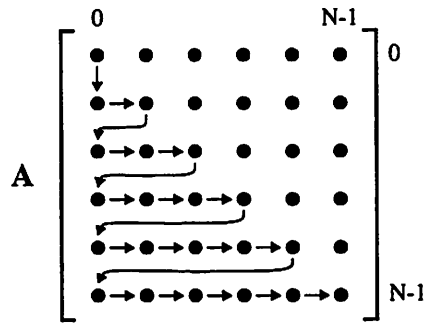

```
p = address_of(A[0]);
for (i = 0; i < N; i++) {
    ....
    p = p + 1;
}
```



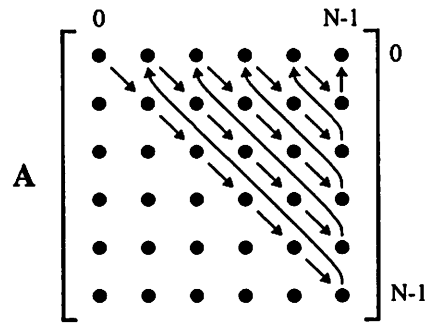

```
p = address_of(A[0]);
for (i = 1; i <= N; i++) {
    for (j = 0; j < i; j++) {
        ....
        p = p + 1;
    }
    p = p - i;
}
```




```
p = address_of(A[0][0]);
for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++) {
        ....
        p = p + 1;
    }
}
```




```
p = address_of(A[0][0]);
for (i = 1; i <= N; i++) {
    for (j = 0; j < i; j++) {
        ....
        p = p + 1;
    }
    p = p + N - i;
}
```




```
p = address_of(A[0][0]);
q = p;
for (i = N; i > 0; i--) {
    for (j = 0; j < i; j++) {
        ....
        p = p + N + 1;
    }
    q = q + 1;
    p = q;
}
```

Note: p and q are address pointers. Arrays are assumed to be stored in row-major format.

Figure 3.6: Examples of Array Access Patterns

signal is limited to frequencies below 4 kHz [30]. The 3-dB bandwidth of the telephone network is approximately 3.6 kHz, and in digital telephony applications, input speech signals are sampled at 8 kHz and coded using Pulse-Code Modulation (PCM) with 8-bit, logarithmic quantization (linear quantization requires 12 bits of resolution for the same level of quality) [31]. This is known as *toll-quality* speech and requires 64 kbit/s of bandwidth to communicate. Speech coders are employed to process this PCM speech signal and reduce the bit rate required to communicate it.

Coders can be broadly classified into two classes: *waveform* coders and *parametric* coders [32]. Waveform coders attempt to reduce the bit rate of the input speech waveform without assuming any knowledge about the nature of speech signals. The simplest waveform coder is the PCM coder with logarithmic quantization, which, as mentioned above, reduces the resolution needed to represent samples of speech signals from 12 bits to 8 bits. Differential Pulse-Code Modulation (DPCM) is an improved coding scheme in which the difference between consecutive samples, as opposed to the actual value of a sample, is transmitted. Since the variance of this difference signal is smaller than that of the original signal, the difference signal can be quantized with fewer bits than the original signal, and the required bit rate is reduced, at the expense of a slight decrease in subjective speech quality. Adaptive Differential Pulse-Code Modulation (ADPCM) is a modified form of DPCM in which adaptive quantization is employed to increase the quality of the resulting speech signal. These coders reduce the required bit rate to 32 kbit/s, and the subjective quality of the speech signal produced by ADPCM, in particular, is very close that of PCM.

Parametric coders use *a priori* knowledge about the nature of speech signals to reduce the bit rate required to communicate them. Parametric coders exploit the fact that speech signals are quasi-stationary in short time intervals of 5-20 ms, during which a sin-

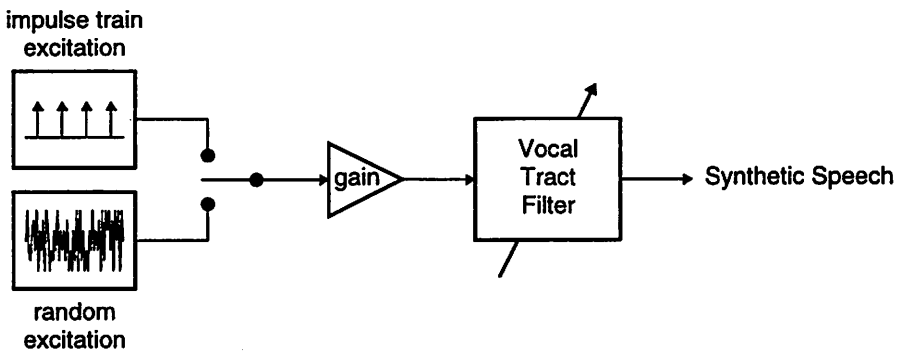


Figure 3.7: Human Speech Generation Model

gle basic sound is being uttered. Samples of the input signal within these time intervals are highly correlated. These correlations can be modeled with a set of parameters that represents the input speech signal during a given interval. The input speech signal is divided into short segments, or *frames*, and a set of parameters representing a whole frame of speech is extracted and transmitted. A close approximation to the original frame of speech can then be reconstructed by decoding the transmitted parameter set.

3.5.1 Speech Generation Model

The speech generation model that is used by parametric speech coders is illustrated in Figure 3.7. This model captures the salient features of the human speech generation process. The sounds of human speech are generated as air from the lungs flows by the vocal cords, and the resulting excitation resonates through the vocal tract, which consists of the cavities of the pharynx, the mouth, and the nose. For *voiced* sounds, which correspond to vowels, the vocal cords vibrate at some pitch frequency, and the resulting periodic excitation, which can be modeled as an impulse train, is shaped by the resonances of the vocal tract. For *unvoiced* sounds, which correspond to consonants, the flow of air is unaffected by the vocal cords, and the resulting excitation, which is the sound of turbulent air flow and can be modeled as a random excitation, propagates through and is shaped by

the vocal tract. The energy of voiced sounds is generally more than the energy of unvoiced sounds.

In the model shown in Figure 3.7, the vocal tract is modeled by a time-varying filter. The vocal tract can be adequately modeled with an N-th order, all-pole filter with the following transfer function [30]

$$H(z) = \frac{1}{1 - A(z)} = \frac{1}{1 - \sum_{i=1}^{N} a_i z^{-i}} \tag{3.5}$$

For each frame of speech, the coefficients of $A(z)$ are determined by Linear Prediction Coding (LPC) analysis [33, 34]. In the LPC framework, $A(z)$ is known as the *short-term predictor.* The value of N depends on the number of resonant modes, or *formants*, of the vocal tract that need to be modeled. Each formant is formed by a complex-conjugate pair of poles, and as there are typically three to five formants below 5 kHz, $N = 10$ is quite adequate for most applications. Depending on the type of sound being generated, either an impulse train, for a voiced sound, or a random excitation, for an unvoiced sound, is selected, weighted by a gain factor, and fed into the vocal tract filter. Since the vocal tract filter synthesizes the speech signal, it is commonly referred to as the *synthesis filter.* In more sophisticated coders, the excitation fed into the synthesis filter is a combination of an impulse train and a random excitation.

A parametric coder based on the model shown in Figure 3.7 analyzes each frame of the input speech signal and extracts the excitation signal, the gain factor, and the coefficients of the synthesis filter for that frame. These parameters are then encoded and transmitted. The decoder at the receiving end uses these parameters to synthesize an approximation to the original frame of speech.

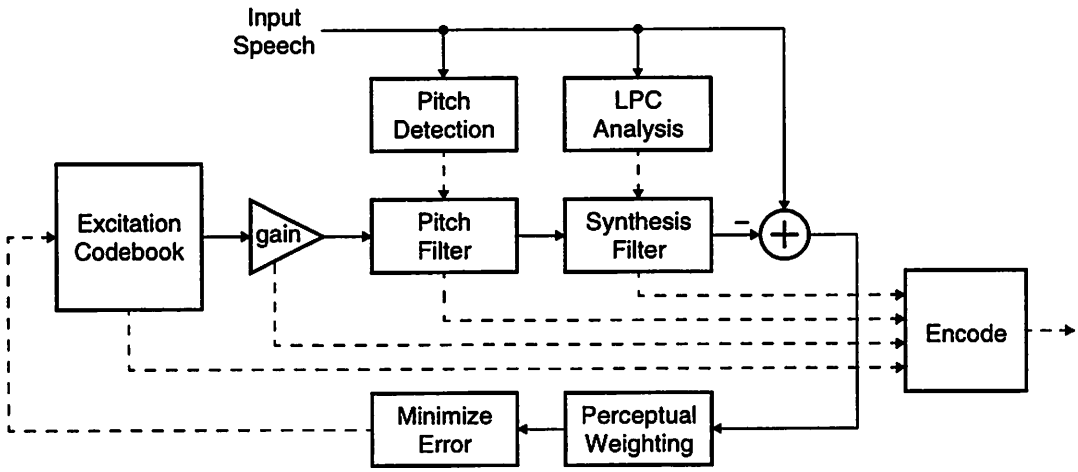


Figure 3.8: Structure of a CELP Speech Coder

3.5.2 Code-Excited Linear Prediction

Almost all modern speech coders are variations of the Code-Excited Linear Prediction (CELP) speech coder [35]. The development of the CELP coder is considered a milestone in speech coding, as it allowed coding of high-quality speech at bit rates below 8 kbit/s. In a CELP coder, the excitation signal is coded using vector quantization [28]. The basic structure of a CELP coder is shown in Figure 3.8. The task of the coder is to select an appropriate excitation vector from a codebook of excitation vectors. This is accomplished by an *analysis-by-synthesis* process [36] during which each vector in the codebook is fed into the synthesis filter, and the synthesized frame of speech is compared to the original frame of input speech. The excitation vector resulting in the least error between the original and synthesized frames of speech is selected. The error criterion is the mean-squared error filtered through a perceptual weighting filter that shapes the spectrum of the quantization noise such that most of the quantization noise energy is concentrated near the spectral peaks of the speech signal where it is largely masked by the human auditory system. The codebook index of the selected excitation vector is transmitted, and

the decoder at the receiving end can select the correct excitation vector from an identical copy of the codebook used by the coder. The coefficients of the synthesis filter are extracted by the coder using LPC analysis. In a CELP coder, vector quantization is used to code the random excitation only. Pitch periodicity of voiced speech is introduced into the excitation by using a pitch filter. The transfer function of the pitch filter is

$$H_L(z) = \frac{1}{1 - A_L(z)} = \frac{1}{1 - \beta z^{-L}} \tag{3.6}$$

where $A_L(z)$ is known as the *long-term predictor*, and L is known as the *lag* and represents the period of the impulse train excitation needed for voiced speech. Once the index of the excitation vector, the coefficients of the synthesis filter, the lag, and the excitation gain for the current frame of the input speech signal are all determined, they are encoded and transmitted.

Searching for the best excitation vector in the codebook is the most time-consuming task in a CELP speech coder. As a result, a great deal of research effort has focused on finding codebook structures that will make the codebook search process more efficient than a straightforward exhaustive search.

3.6 Vector-Sum Excited Linear Prediction

The Vector-Sum Excited Linear Prediction (VSELP) algorithm was developed for use in cellular and mobile telephony applications [37]. An 8-kbit/s VSELP speech coder was adopted for the IS-54 North American Digital Cellular standard. The VSELP algorithm requires about 15 MIPS of computational performance.

The basic structure of the VSELP coder is shown in Figure 3.9. The 8 kHz input speech signal is divided into 20-ms frames (160 samples). Each frame is sub-divided into

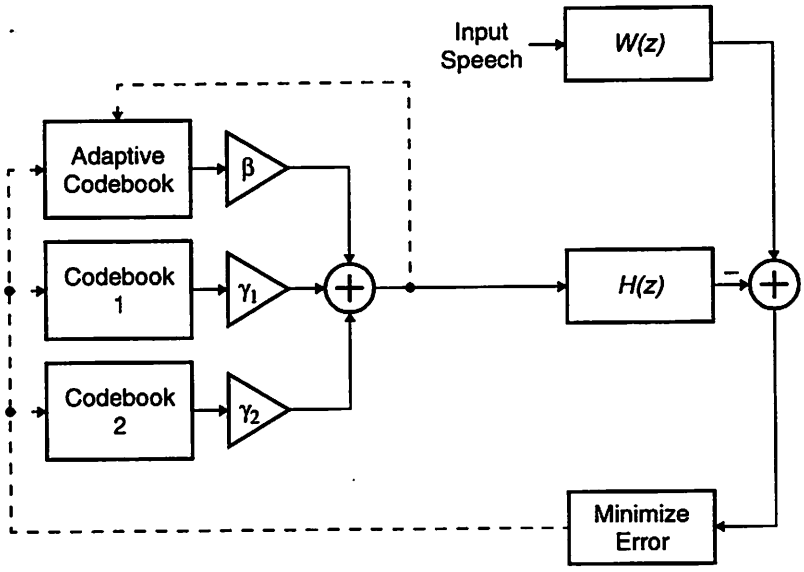


Figure 3.9: Basic Structure of the VSELP Coder

5-ms subframes (40 samples) that are processed independently. A 10-th order synthesis filter of the form shown in Equation 3.5 is used. The synthesis filter is combined with a perceptual weighting filter to form a weighted synthesis filter. The transfer function of the perceptual weighting filter is

$$W(z) = \frac{1 - A(z)}{1 - A(z/\lambda)} \tag{3.7}$$

where $A(z)$ is the short-term predictor of the synthesis filter, and $\lambda = 0.8$. The transfer function of the weighted synthesis filter is thus

$$H(z) = \frac{1}{1 - A(z/\lambda)} = \frac{1}{1 - \sum_{i=1}^{N} \frac{a_i}{\lambda} z^{-i}} \tag{3.8}$$

For each frame, the coefficients of $H(z)$ are determined by LPC analysis. These coefficients are used in the fourth subframe. The coefficients used in the other three subframes

are computed by linearly interpolating the coefficients of the fourth subframe of the previous and current frames.

The excitation vector is derived by combining vectors from three separate codebooks: an adaptive pitch codebook and two stochastic codebooks. The criterion for selecting an excitation vector $\mathbf{u} = \langle u[0], \ldots, u[N-1] \rangle$ ($N = 40$ is the subframe length) is to maximize C^2/G, where

$$C = \sum_{n=0}^{N-1} u'[n]\, p[n] \tag{3.9}$$

$$G = \sum_{n=0}^{N-1} (u'[n])^2 \tag{3.10}$$

$u'[n]$ is the filtered code vector and $p[n]$ is the perceptually weighted input speech vector. This is equivalent to minimizing the mean-squared error.

The pitch codebook is used to implement the pitch filter of Equation 3.6. The pitch codebook is adaptive, and it implements the functionality of the delay line associated with the z^{-L} term of the pitch filter. The codebook stores the past 146 samples of the excitation signal. Each value of the long-term prediction lag L corresponds to a subframe of excitation from the past starting L samples ago (see Figure 3.10). L can range from 20 to 146 (127 codes). The 128-th code for L is used to indicate that the pitch codebook is not to be used. When $L < 40$, the period of the excitation pitch is less than the length of a subframe, and the amount of history in the adaptive codebook is not sufficient to construct a full subframe of excitation; therefore, the available history is repeated to create a full subframe of excitation. After all codebooks have been searched, and the excitation for the current subframe is completely determined, the adaptive codebook is updated.

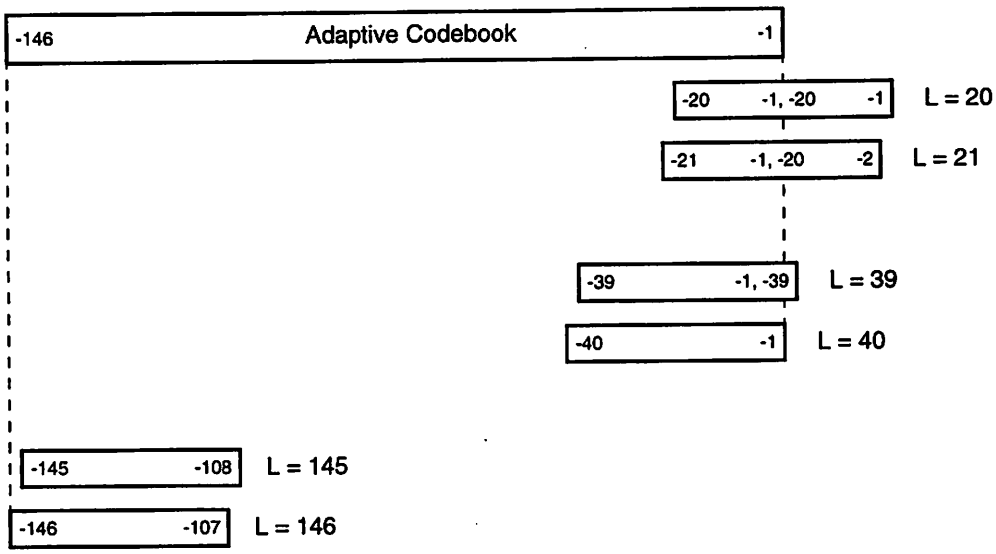


Figure 3.10: Structure of the Adaptive Pitch Codebook in VSELP

The stochastic codebooks in VSELP are highly structured and can be searched efficiently. Each codebook consists of 128 code vectors $\mathbf{u}_i$ $(0 \le i \le 127)$. These vectors are different linear combinations of seven basis vectors $\mathbf{v}_m$ $(1 \le m \le 7)$:

$$\mathbf{u}_i = \sum_{m=1}^{7} \theta_{i,m} \mathbf{v}_m \tag{3.11}$$

where $\theta_{i,m}$ can be either +1, if the m-th bit of the code index is 1, or -1, if the m-th bit of the code index is 0. This scheme greatly simplifies the codebook search process because the response of the weighted synthesis filter to each code vector can be obtained by combining the filtered basis vectors, instead of filtering the code vectors. In addition, the effect of changing one bit in the code word, due to a transmission error, for instance, is not catastrophic, as the erroneous vector is different from the correct one only by one basis vector.

The codebooks are searched sequentially. First, the pitch codebook is searched. Next, the basis vectors of the first stochastic codebook are orthogonalized to the filtered

excitation vector from the pitch codebook using the Gram-Schmidt approach [38]. The orthogonalized basis vectors are then filtered, and the codebook is searched. Next, the basis vectors of the second stochastic codebook are orthogonalized to the filtered excitation vectors from the first two codebooks. The orthogonalized basis vectors are then filtered through the weighted synthesis filter, and the second codebook is searched. The codebook gain factors β, γ_1, and γ_2 are determined during the search process, and are then jointly quantized using a vector quantizer. Once the excitation vector for the current subframe is completely determined, the adaptive codebook is updated such that the new excitation vector becomes the most recent history in codebook. Further implementation details of the VSELP algorithm can be found in the IS-54 standard description [39].

3.6.1 Analysis of the VSELP Algorithm

The execution profile of the VSELP algorithm is shown in Table 3.1. The table shows the percentage of total execution time for the most time-consuming functions in VSELP. The data in this table is based on an implementation of the VSELP algorithm in the C programming language running on a Sun SPARC processor. The profile of the program was obtained using a run-time software profiler. As in all CELP coders, most of the execution time of the VSELP algorithm is spent on searching the codebooks for the best excitation vectors.

We can gain more insight into the computational complexity of the VSELP algorithm by looking at the execution profile of the dominant kernels. Table 3.2 shows the execution profile of the four most dominant kernels of the VSELP algorithm. These four kernels account for 85% of the total execution time of the VSELP algorithm. Furthermore, the two most dominant kernels account for 76% of the total execution time. Thus, an efficient implementation of the VSELP algorithm will require that these four kernels be executed very efficiently, with a minimum of time and energy overhead.

3.7 Algorithm Domains

The CELP coder is the prototype on which almost all modern speech coders are based. Many different CELP-based algorithms have been developed for voice communication applications, especially mobile and cellular telephony. Some examples include: VSELP [37], DoD CELP [40], LD-CELP [41], PSI-CELP [42], ACELP [43], and CS-

Function	*% Time*
`FilterCodebook()`	46.5
`ComputeLag()`	21.9
`CodebookSearch()`	8.2
`ComputeWeightedInputSpeech()`	3.9
`IIRFilter()`	3.7
`QuantizeGains()`	3.5
`OrthogonalizeCodebook()`	2.8
`MatrixMultiply()`	2.6
`LPCAnalysis()`	2.1
`StateAdvanceToTime()`	1.8
`UpdateFilterState()`	1.7

Table 3.1: Execution Profile of the VSELP Algorithm

Kernels	*% Time*
`WeightedSynthesisFilter()`	45.6
`DotProduct()`	30.5
`IIRFilter()`	7.2
`FIRFilter()`	1.2

Table 3.2: Dominant Kernels in the VSELP Algorithm

ACELP [44], to name just a few. Collectively, these different algorithms form a *domain* of algorithms, as they have some basic similarities. Algorithms within a domain have similar computational structures, dominant kernels, data structures, and word-lengths. Differences among algorithms within a domain are mainly due to the values of the basic parameters and the high-level control flow of each individual algorithm. CELP-based coders, for example, use the same basic analysis-by-synthesis computational structure consisting of codebooks and speech synthesis filters modeling the human vocal tract, they all use 16-bit arithmetic, and they spend most of their execution time computing vector dot products, filtering code vectors, and synthesizing speech frames with different excitation vectors. Differences among these algorithms are mainly due to differences in the structure and the number of codebooks that are used, and in the parameters of the synthesis filter such as the number of filter taps and the resolution of filter coefficients.

Another example of a DSP algorithm domain is formed by video compression/decompression algorithms that are based on the Discrete Cosine Transform (DCT) [45]. There are a number of different algorithms and standards that are in wide-spread use, such as H.261, MPEG, MPEG-2, and MPEG-4 [46]. All of these algorithms are based on DCT, Inverse-DCT, and motion vector estimation/compensation. They vary in the high-level control flow and in the value of the basic algorithm parameters, but they can be implemented using similar hardware structures [47].

Because of their underlying similarities, algorithms within a domain can be implemented using similar hardware architectures. By executing the dominant kernels of a given domain of algorithms on highly optimized processing units that incur minimal energy and performance overhead, we can build processors that are highly efficient. Processors of this type are known as *domain-specific* processors. The work presented in this dissertation was focused on designing energy-efficient domain-specific processors.

3.8 Architectural Requirements for Digital Signal Processing

We end this chapter with a list of architectural requirements that must be satisfied in an efficient programmable architecture for digital signal processing algorithms:

- DSP algorithms exhibit high levels of temporal and spatial concurrency. A programmable DSP architecture must be able take advantage of this concurrency and support pipelined and parallel modes of processing efficiently.
- Concurrent processing increases the required instruction and data memory memory bandwidth. The memory structure of a programmable architecture must be able to support the increased memory bandwidth requirement efficiently without incurring significant delay and energy overhead.
- The interconnection network that links the memories and the processing elements must support high data rates and must be flexible enough to support the required communication patterns that are commonly seen in DSP kernels.
- The control structure that is used to coordinate computational activities within multiple parallel processors and memories must be efficient and scalable.

CHAPTER 4

Programmable Architectures for Digital Signal Processing

In this chapter we will take a broad look at some of the basic approaches that have been taken in designing programmable architectures for digital signal processing applications. Our goal will be to characterize and differentiate these approaches and to develop an understanding of their strengths and weaknesses. The main focus of this analysis will be the energy efficiency of these architectural approaches. Our discussion will lead to a set of architectural design principles for energy-efficient programmable signal processors. These principles form the basis of the design choices made in the Pleiades architecture template.

4.1 Basic Model for Programmable Hardware

In order to characterize and classify programmable processor architectures, we first need to have a basic model that captures the essence of programmable computing. Our concern here is with the structure of programmable computing devices, so it is natural to focus on the fundamental components that all such devices consist of and on the interactions between those components. A basic model for programmable processors is illustrated in Figure 4.1. No particular hardware organization is to be inferred from this

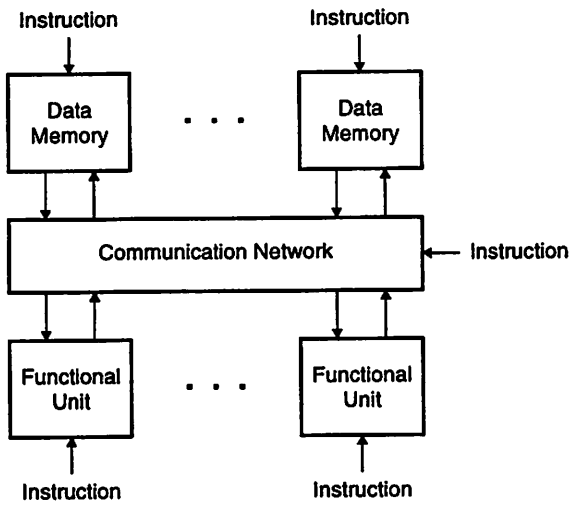


Figure 4.1: Basic Model for Programmable Processors

illustration, which is merely an abstraction of the basic components of a programmable processor and the interactions among those components. All programmable processors have the following basic components:

- *Functional units* that can perform the various arithmetic and logic functions that are required by the computations that a processor is expected to perform.
- *Memory units* to store the data operands processed by the functional units.
- A *communication network* that allows the exchange of data between the functional units and the memory units.
- *Instructions* that control the actions taken during each execution cycle by the above components.

The distinguishing feature of programmable processors are the instructions. The ability to perform different tasks at different times under the control of instruction sequences is what makes a computing device programmable and gives it the flexibility to perform different computations. During each execution cycle, instructions control what

data operands are read from the memory units, how these operands are routed to the functional units, what types of operations are performed by the functional units on the operands provided to them, how results produced by the functional units are routed back to the memory units, and where in the memory units these results are stored. Instructions require two other basic components that are present in all programmable processors:

- *Instruction memory* where instructions can be stored.
- An *instruction control mechanism* that coordinates the delivery of instructions from the instruction memory to the functional units, the data memory units, and the communication network.

Differences among programmable architectures are due to the organization of the hardware resources that are used to implement the basic components outlined above, and the task of an architect is to organize hardware resources in such a way that the resulting processor can perform the required computational tasks efficiently. The variety of ways in which hardware resources can be organized is virtually limitless, and an architect has a number of important issues and design parameters to settle:

- Functional units - A key architectural issue is the *granularity* of the functional units. Granularity is a measure of the complexity of the operands processed by the functional units (e.g., bits, integers, floating-point numbers, vectors) and the complexity of the instructions executed by the functional units. A related issue is the variety of instruction types executed by the functional units. An important performance parameter is the number of functional units available in a processor, as it determines the number of computations performed in each cycle by the processor.
- Data memories - The bandwidth of the data memories, i.e., the number of read and write operations that can be performed in each execution cycle, is an important performance parameter. It is a function of the organization of the data memory.

The size of the data memory is also an important design parameter, as the amount of data that can be stored in memory determines the complexity of the algorithms that can be executed by a processor.

- Communication network - The bandwidth and the flexibility of the communication network are important design considerations. Bandwidth refers to the number of operands that can be transferred through the communication network in each cycle. Flexibility refers to the richness of the communication patterns that can be supported by the communication network.
- Instructions - The organization of the instruction memory and the instruction control mechanism is one of the most important design issues in a programmable architecture and has a strong influence on how efficient an architecture can be. The bandwidth of the instruction memory is one of the key performance parameters of a processor, as it determines the number of instructions that can be executed in each cycle. Another important parameter is the depth of the instruction memory, i.e., the length of the longest sequence of instructions that can be stored in the instruction memory, which is a measure of the complexity of the algorithms that can be executed by a processor. Another key architectural parameter related to instructions is the number of control threads that are used in the instruction control mechanism.

The design space for programmable architectures is defined by the parameters outlined above. These parameters (or subsets thereof) can also be used to classify programmable architectures. Flynn, for example, has proposed a simple taxonomy based on the number of threads in the instruction control mechanism and the number of functional units [48]. In Flynn's taxonomy, there are three basic processor types: SISD (single instruction, single data), SIMD (single instruction, multiple data), and MIMD (multiple instruction, multiple data). Skillicorn extended Flynn's taxonomy to include details of the interconnec-

tion networks from the functional units to the data and instruction memories [49]. DeHon classifies processor architectures using a basic architectural model in which instructions are dispatched only to functional units that have local storage for data (there are no independent data memories) and can communicate with one another via an interconnect network [50]. Architectures are classified by four parameters: number of control threads in the instruction control mechanism, number of instructions (same as number of functional units in DeHon's scheme) per control thread, depth of the instruction memory, and granularity of the functional units. DeHon's notion of granularity refers only to the complexity of the operands processed by the functional units. DeHon's scheme is more useful than Flynn's and Skillicorn's in evaluating the merits of an architecture, as it considers more details and is more quantitative in its approach. Our discussion of the energy efficiency of different architectural approaches will be in terms of the basic architectural model shown in Figure 4.1, and we will consider all relevant architectural parameters, as necessary.

4.2 Energy Consumption in Programmable Architectures

In assessing the energy efficiency of programmable architectures, it is important to know what the basic components of energy consumption are when algorithms are implemented on programmable architectures. It can then be determined which components are fundamental and cannot be avoided and which components are not fundamental and must be minimized.

The division of total energy consumption into basic components can be done in terms of the basic hardware components that were outlined in the last section. The basic components of energy consumption of an algorithm implemented on a programmable architecture are due to:

- Computation of the basic arithmetic and logic functions required by an algorithm using the functional units.

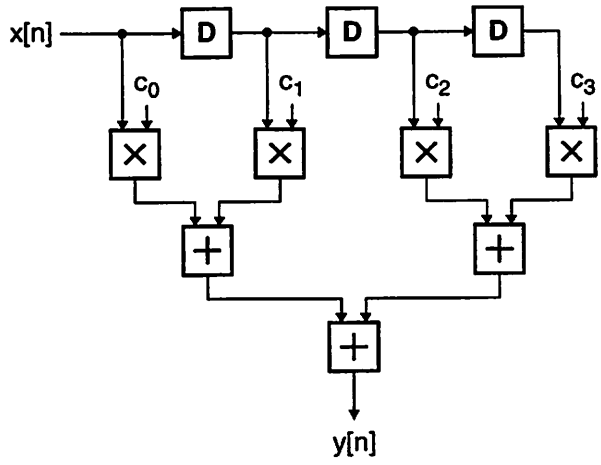


Figure 4.2: Custom Implementation of a 4-Tap FIR Filter

- Storage and access of variables in the data memories.
- Communication of operands among the functional units and the data memories.
- Control of computation, storage/access, and communication activities through instructions.

In order to determine which components of energy consumption are fundamental and which ones are not, and to gain a better understanding of the causes of inefficiencies in programmable architectures, it is instructive to consider the components of energy consumption in custom, application-specific implementations. In a custom implementation, the properties of a given algorithm can be freely exploited to create an optimized implementation than consumes minimal energy. As a result, a custom implementation can be used as a reference to which other implementations based on programmable architectures can be compared in order to evaluate their energy efficiency. Figure 4.2 shows the block diagram of a custom, application-specific implementation of a 4-tap FIR filter. In this custom design, the hardware blocks that are used are not any larger or more complicated than they need to be. Each hardware block performs a specific task (e.g., multiply, add, delay)

and consumes only the basic minimum energy required to perform that task. The word-lengths of the registers, adder, multipliers, and buses do not have to be any larger than the required minimum. The energy consumed by the hardware blocks used in this implementation is due to the basic computations of the algorithm and the storage/access of the state variables of the algorithm. Since each hardware block performs a specific function, there is no need for instructions, and the energy overhead of delivering instructions to the hardware blocks is avoided. In addition, since the locality of reference particular to this algorithm can be preserved using custom placement of the hardware blocks, the energy of communicating data operands is minimal. Notice that in a time-multiplexed implementation, we would have to store and access intermediate variables, and we would also need a controller to instruct the hardware resources to perform the basic computational steps in the proper sequence. Thus, a time-multiplexed design introduces an energy overhead that is not present in a direct implementation. As a result, energy consumption due to storage/access of intermediate variables and due to time-multiplexed control is not fundamental and should be minimized. This must be balanced against the area advantage of a time-multiplexed design.

In summary, energy consumption due to basic computations and storage/access of state variables can be considered fundamental, and energy consumption due to communication, storage/access of intermediate variables, and control is overhead and must be minimized. In the following sections, we will consider different programmable architectures, and we will discuss their strengths and weaknesses in terms of the energy overhead they incur.

4.3 General-Purpose Processors

Figure 4.3 shows the basic architectural model for a general-purpose processor. There is a single functional unit[1] that can compute a wide variety of arithmetic and logic

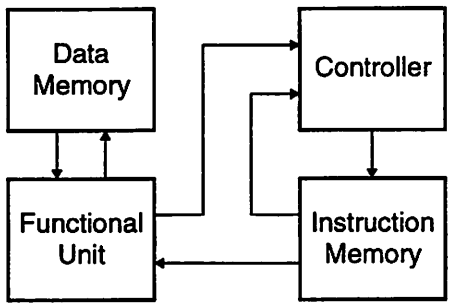


Figure 4.3: Basic Architectural Model of a General-Purpose Processor

functions for n-bit operands, and there is a single data memory where data operands are stored. The communication network is a simple n-bit bus connecting the data memory to the functional unit. Instructions are fetched from the instruction memory and delivered to the functional unit and the data memory by a simple control mechanism that has a single thread of control. The controller makes its decisions based on control instructions from the instruction memory and results of the computations performed by the functional unit. Typically, the instructions stored in memory are encoded to take up less space, and they need to be decoded before they are delivered to their destinations. This type of hardware organization is commonly known as the von Neumann architecture, as it is commonly attributed to John von Neumann [51].

General-purpose processors represent the ultimate in flexibility, as they can be programmed to implement any algorithm. This flexibility is, however, achieved at a significant cost compared to application-specific devices. The energy overhead of implementing an algorithm as a program running on a general-purpose processor is significant. Every single computational step, e.g., addition of two numbers, requires fetching and decoding an instruction from the instruction memory, accessing the required operands from the data memory, and executing the specified computation on a general-purpose functional unit

1. We will discuss variations of this baseline architecture with more than a single functional unit in the following sections.

that is designed to perform a wide variety of computations. All of these activities involve accessing large, centralized memories, performing calculations on large, complex datapaths, and driving long, heavily-loaded wires, and as a result, a great deal of energy is consumed. If the bit-width of the functional unit is larger than the word-lengths used in the algorithm, then additional energy is wasted. Another weakness of general-purpose processors is that computations are done in highly time-multiplexed fashion. To achieve high performance, a general-purpose processor must run at a high clock frequency; therefore, the supply voltage cannot be aggressively reduced to save energy. In addition, the amount of switching activity is increased as temporal correlations that are especially common in signal processing applications are not preserved.

There are a number of techniques that can be used to improve the energy efficiency of general-purpose processors. Introducing hierarchy into the memory structures is a technique that was originally introduced to improve the performance of general-purpose processors [52], but it can also reduce the energy of memory accesses. At the lowest level of the data memory hierarchy in modern general-purpose processors is a register file, where scalar and temporary variables are stored. The register file is usually part of the functional unit datapath. Since the register file is small and physically close to the functional unit, it requires much less energy than the main data memory to store and access data operands. Next in the hierarchy is a data cache, which stores the most recently used operands. Most data access requests are satisfied by the data cache, which is smaller than the main memory and consumes less energy to access than the main memory, which is at the top of the data memory hierarchy. An instruction cache is also used to reduce the overhead of fetching instructions from the instruction memory. In some architectures, the instruction and data caches are merged into a unified cache structure [53]. Almost all modern general-purpose processors execute instructions in a pipelined fashion whereby instruction fetch, instruction decode, operand access, instruction execution, and result write-back steps of a

few consecutive instructions can be performed concurrently. The resulting increase in performance relaxes the need to increase the clock frequency and can be traded off to reduce power by reducing the supply voltage. To minimize unnecessary switching activity, many modern microprocessors use power-down modes and clock-gating techniques that allow shutting down unused circuit modules [54, 55]. Another technique that has been applied to reduce the energy overhead of the instructions is to use instruction formats and addressing modes that require smaller number of bits to encode [56, 57]. This reduces the bit-width of the instruction memory and the instruction bus and reduces the energy overhead of fetching instructions.

While the techniques mentioned above are useful in improving the energy efficiency of general-purpose processors, the fact remains that programmed implementations of DSP algorithms on general-purpose programmable architectures are far too inefficient compared to custom implementations. For example, the custom FIR filter shown in Figure 4.2, designed for 16-bit input samples and coefficients, consumes 155 pJ of energy per tap, when implemented in a 0.6-μm CMOS process, with a supply voltage of 1.5 V. When normalized to the same process and supply voltage used for the custom design[1], the energy consumed by the StrongARM microprocessor [58, 59], which is highly optimized for low-power operation, is 37.4 nJ per tap, i.e., 240 times more than the custom design! The maximum sample rate of the custom design is 18 MHz, whereas the maximum sample rate of the StrongARM implementation is 532 kHz. More detailed energy and performance comparisons will be presented in Chapter 7.

4.4 Programmable Digital Signal Processors

Programmable digital signal processors are similar to general-purpose processors, but they are optimized for signal processing algorithms. The basic architectural model of

1. See Chapter 7 for details of the normalization procedure.

general-purpose processors shown in Figure 4.3 is also valid for programmable signal processors. As a result, programmable signal processors suffer from the same overheads and inefficiencies that general-purpose processors do, but a number of architectural improvements make them far more efficient than general-purpose processors.

When programmable processors were first introduced [60, 61], one of the key features that differentiated them from general-purpose processors and made them more suitable for DSP algorithms was hardware support for fast, i.e., single-cycle, multiplication. This capability is particularly useful, as DSP algorithms tend to use multiplications very frequently, and the ability to perform multiplications at a high rate provides for significant speed-up compared to shift-and-add software routines that are commonly used in general-purpose processors. All modern DSP processors can perform a multiply-accumulate (MAC) operation, which is very common in DSP algorithms, in a single execution cycle. They also use large accumulators that allow them to add a large number of products before overflowing.

In addition to fast multiplication, DSP processors have also relied on concurrent processing to improve performance. Instructions are typically executed in a pipelined fashion. This allows the processor to overlap the execution of a few consecutive instructions. The depth of the instruction execution pipeline has increased in modern DSP processors, but branch instructions limit the amount of speed-up that can be achieved by increasing the depth of the instruction execution pipeline. Another form of concurrency that is common in DSP processors is the ability of the multiplier unit to operate in parallel with the arithmetic/logic unit [62, 63]. The data memory bandwidth has to be increased, as well, if the parallelism in the functional unit is to be exploited. Thus, DSP processors tend to use multiple memory banks that can be accessed in parallel. In its simplest form, the result is the Harvard memory architecture introduced in the Texas Instruments TMS32010

processor [61], where the instruction memory and the data memory were split into separate physical entities and could be accessed simultaneously. This allows the processor to access a data sample and a coefficient simultaneously and improves the performance of FIR and IIR filters, for example. Modern DSP processors typically allow simultaneous access for an instruction and two data operands from parallel memory banks [64, 65].

Another innovation in DSP processors was to introduce memory addressing modes that allowed the calculations of memory addresses to be done in parallel with data calculations. One particularly useful addressing mode is the register-indirect with post-increment addressing mode that allows one to sequence through an array by incrementing the address pointer automatically without the need to execute a separate addition instruction for that purpose. Another useful addressing mode for implementing FIR filters, for example, is the circular addressing mode that allows the address pointer to wrap around to the beginning of a memory address block. The required bound check is executed in parallel without taking an extra processing cycle. All modern DSP processors have hardware support for zero-overhead looping. This is typically done with a repeat instruction that allows the repetitive execution of a small sequence of instructions without taking any extra processing cycles for loop index calculations (i.e., index increment/decrement and bound check). Combined with concurrent memory accesses, this allows DSP processors to perform an n-element vector dot product in $n + 1$ cycles, and as a result, modern DSP processors are particularly well-suited for implementing vector dot products and FIR filters. Another technique that is used by some DSP processors is to provide hardware support to execute complex instructions. One good example of this is hardware support for the Viterbi add-compare-select calculations which has been used by a number of DSP processors geared to cellular communications applications [64, 65]. Another architectural improvement in some of the latest DSP processors is to add an extra multiplier and adder to increase the processing rate for FIR filters [66, 67]. This is typically done without increas-

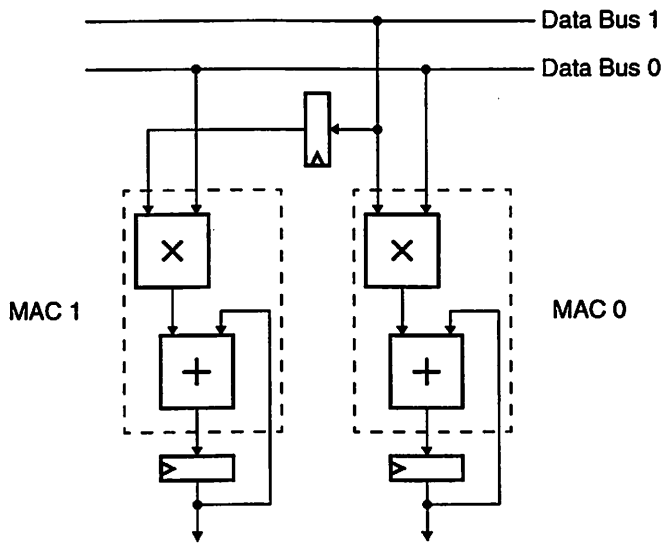


Figure 4.4: Dual-MAC Structure of the TCSI LODE Processor

ing the data memory bandwidth to the functional unit, and as a result, the extra multiplier and adder blocks can be used in limited ways that rely on locally stored operands that were read from memory during the previous cycle (see Figure 4.4). This arrangement is useful for improving the performance of FIR and IIR filters. The LODE processor from TCSI is a good example of how this can be done [67]. Additional hardware resources are typically utilized in DSP processors by adding new instructions that encode more operations into the basic instruction format. While this factor minimizes the instruction overhead of the additional hardware resources, it does make these processors difficult to program and difficult to generate code for. As a result, DSP processors must be programmed in assembly language to achieve good performance.

Another innovation that can be used to reduce the energy overhead of instructions while executing loops is to use decoded instruction buffers [68]. In this technique, when the body of a loop is executed for the first time, the decoded instruction sequence (i.e., the control signals derived by decoding instructions) corresponding to the loop body is cap-

tured into a small local buffer, i.e., the decoded instruction buffer. Subsequent iterations of the loop use these decoded instructions from the buffer instead of fetching and decoding instructions from the instruction memory. In this way, the larger energy overhead of accessing the instruction memory through the instruction bus and decoding of fetched instructions is replaced by the smaller energy of accessing the decoded instruction buffer. The reported energy savings is on the order of 40%.

All of these architectural techniques have helped make programmable signal processors much more efficient than general-purpose processors at performing some of the most common DSP calculations such as FIR and IIR filters. When normalized to the same 0.6-μm process and 1.5-V supply voltage used for the custom design shown in Figure 4.2, the energy consumed by the Texas Instruments TMS320C54 DSP processor [64], which is highly optimized for low-power operation, is 0.6 nJ per tap, and the maximum sample rate is 5.1 MHz. This is a significant improvement over the StrongARM processor. This should be no surprise, however, as modern DSP processors are particularly well-optimized for one-tap-per-cycle FIR and IIR filter implementations. However, they do not do nearly as well for other signal processing algorithms, such as the FFT, compared to custom implementations.

4.5 Superscalar and VLIW Processors

The basic idea behind superscalar and VLIW (very long instruction word) processors is to improve the performance of the basic von Neumann architecture by adding more functional units to execute more instructions in parallel. Superscalar and VLIW processors are very similar to each other in this respect.

The basic architectural model for superscalar and VLIW processors is shown in Figure 4.5. Instead of a single functional unit, there are multiple functional units that can

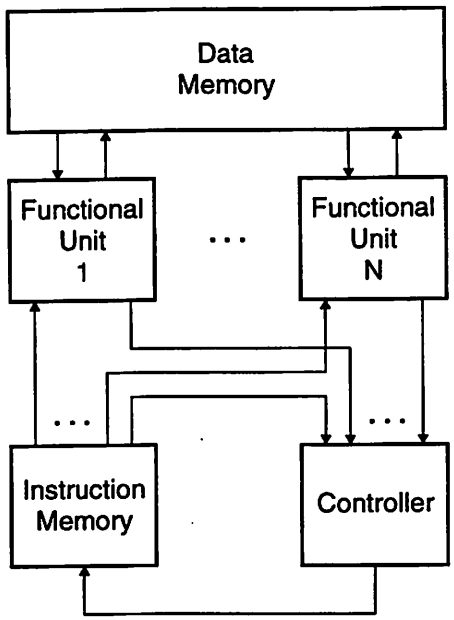


Figure 4.5: Basic Architectural Model for Superscalar and VLIW Processors

operate in parallel. The instruction memory must, therefore, issue multiple instructions. To ensure that the functional units can be supplied with data operands adequately, the bandwidth of the data memory must be increased. This is typically done by adding multiple read and write ports to the register file (the lowest level of the data memory hierarchy). In some implementations, the data cache can supply multiple operands per execution cycle [69].

Superscalar and VLIW processors are designed to take advantage of fine-grain parallelism. This is the type of parallelism that exists within a basic block, i.e., a maximal sequence of instructions ending in a control transfer instruction, e.g., branch and subroutine call instructions. In scalar programs the amount of parallelism is application-dependent and is typically not very high, so the degree of performance enhancement obtained by these processors is limited to a factor of approximately four [70]. Vector processing algorithms, and in particular DSP algorithms, have lots of coarse-grain parallelism, i.e., parallelism across multiple iterations of a loop, which can be exploited by VLIW and

superscalar processors, but the main bottleneck for vector programs is the data memory bandwidth. The multi-port register file at the lowest level of the memory hierarchy can provide good performance for fine-grain parallelism, but to improve performance for vector programs, the data memory bandwidth must be increased.

The chief difference between superscalar and VLIW processors is in the manner in which instructions are issued. A superscalar processor fetches a block of instructions from the instruction memory in parallel, and the decoding hardware analyzes the data dependencies between the fetched instructions. For each execution cycle, this analysis results in a set of instructions that have no data dependencies and can be executed in parallel. These instructions are then issued to and executed by the appropriate functional units in parallel. The instruction decoder in superscalar processors is, thus, highly complex, as it has to analyze data dependencies among the instructions fetched from memory, schedule the execution of instructions, and then assign them to functional units. The result is a great deal of design complexity and energy overhead. The primary benefit of superscalar processors is that they can execute available executable binary codes without the need to recompile. This is a tremendous advantage for general-purpose applications, but it is much less of an issue for DSP applications, where programs are typically written in assembly language to optimize performance, anyway, and there is much less of a need to run pre-compiled shrink-wrapped software packages.

VLIW processors, on the other hand, expose the internal micro-architecture of the processor to the compiler. Data dependency analysis, instruction scheduling, and allocation are all done at compile-time. The code generated for the processor consists of long instruction words (hence the name) that contain multiple instruction fields for each functional unit. These instruction words are fetched by the instruction decoder and each field is decoded independently and issued to the corresponding functional unit. As a result, the

decoder in a VLIW processor is much simpler than that of a superscalar processor, and VLIW processors are in general more energy efficient than superscalar processors because they avoid the energy overhead of the vastly more complex decoder of the superscalar processors. The chief drawback of VLIW processors is that they can not be binary-compatible with previous processor generations, as each instance of a VLIW processor has its own long instruction word format. The other drawback of VLIW processors is that if there are not enough instructions in a given cycle to keep all functional units busy, then memory bandwidth and energy is wasted by empty instruction fields. Some recent VLIW architectures have reduced this penalty by encoding instructions in a way that empty instruction fields are not created. This requires a more complex decoder to extract the instruction fields for the current execution cycle from the long instructions fields fetched in the current and possibly the previous fetch cycle.

The other difficulty for VLIW processors is that they are difficult to generate good code for, and as a result, to get good performance programs must be written in assembly language. This is a difficult task, as the activities of multiple functional units must be scheduled and coordinated by the programmer.

Modern high-performance DSP processors are for the most part based on the VLIW scheme[1]. The TMS320C6X processors from Texas Instruments, for example, can issue up to eight instructions in each cycle to six ALUs and two multipliers [71]. The simplicity of the VLIW scheme helps reduce the energy overhead of instructions, as the complex decoding logic needed by superscalar processors is avoided. Still, both of these schemes suffer from the underlying problem of all general-purpose processors, and that is the tremendous energy overhead of fetching and decoding instructions and accessing large, centralized hardware resources.

1. The one notable exception is the superscalar signal processor from ZSP [72].

4.6 Pipelined Vector Architectures

Vector processing architectures were originally developed for scientific applications with massive computational demands, such as problems in nuclear physics, weather forecasting, and seismology [73, 74]. These applications deal with vector data types, i.e., arrays of numbers, and as a result, the processor architectures that were developed for handling these applications have been known as vector architectures. In vector algorithms, a given computation is repeated on different elements of the input vector operands. There is typically no or very little dependency between different iterations of the loop processing the input vectors, and as a result different iterations of a loop can be executed either completely in parallel or in a highly overlapped fashion. As was shown in Chapter 3, signal processing algorithms fall into the class of vector algorithms, and as a result, vector architectures are indeed suitable for implementing DSP algorithms. It should be noted that vector processors are always designed in the form of a vector execution unit coupled to a conventional scalar unit, as there is always some scalar processing and flow control for which the vector units are not suitable. The basic architectural model for vector processors is shown in Figure 4.6.

Vector architectures take advantage of the properties of vector algorithms by introducing vector instructions that deal with vector variables, as opposed to scalar variables. For example, instead of executing many scalar add instructions within the body of a loop and instructions for loop index calculations to perform vector addition, as is done on von Neumann architectures, in a vector processor, a vector add instruction (e.g., VADD) is executed instead. Thus, a whole scalar loop is replaced by a single vector instruction. Since the scalar additions that a vector addition is made of are independent of each other, the additions can be done in a highly pipelined fashion allowing for very high clock speeds. Thus, vector architectures increase performance by allowing very deep, high-

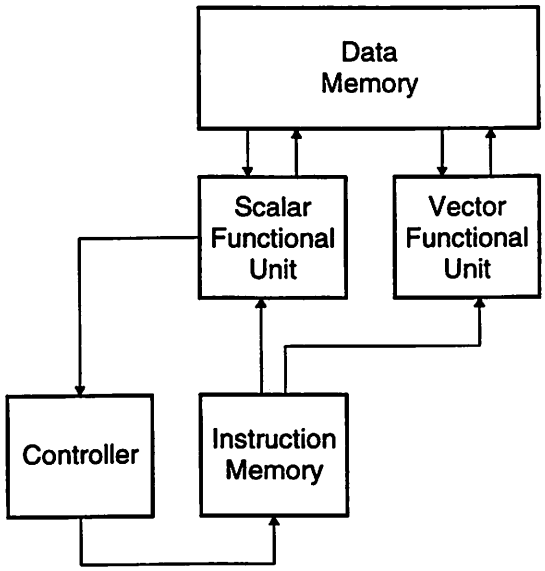


Figure 4.6: Basic Architectural Model for Vector Processors

speed arithmetic pipelines that execute vector arithmetic instructions at a very high rate. In order to achieve high performance on a vector architecture, a given algorithm must be amenable to being coded with vector instructions, i.e., the algorithm has to be vecotrizable. If an algorithm is vectorizable, then powerful compiler techniques exist that can produce high quality vector code from a high-level language specification of the algorithm [75].

Vector architectures are attractive from the point of view of energy efficiency. The reason for this is that vector instructions can significantly reduce the energy overhead of fetching, decoding, and issuing instructions. Instead of fetching multiple instructions for each iteration of a loop for loop index and data calculations, a single vector instruction is fetched and issued, and most of the energy is spent on executing the instruction. Of course the energy overhead of accessing centralized data memories and vector register files and executing instructions on general-purpose arithmetic pipelines still remains, but much less energy is wasted on fetching and decoding instructions. As a result, researchers have

machine is the structure of the communication network that connects the functional units within the SIMD array and the data memory, and numerous interconnection network topologies, such as mesh, hierarchical mesh, and hypercube, have been proposed [74]. One of the key issues in programming SIMD machines is to map vector operations onto the SIMD array given the constraints of the interconnection network being used. As a result, SIMD machines are generally difficult to program.

The advantage of SIMD architectures is that they can achieve high performance without incurring a large increase in the instruction bandwidth. The energy overhead of fetching an instruction is reduced because a fetched instruction is used by all of the functional units in the SIMD array. However, SIMD architectures incur the overhead of broadcasting the single instruction to all functional units. In addition there is the energy overhead of the interconnection network that further complicates the design space.

In some recent microprocessors and DSP processors, SIMD instructions have been used as a set of multimedia extensions to the basic instruction set of the processor in order to improve the performance of the processor for multimedia applications, e.g., video decompression [78]. These instructions use the wide ALUs of modern processors to execute multiple low-resolution operands in parallel. For example, a 32-bit ALU is used to perform arithmetic operation on four 8-bit operands in parallel. This extension involves a minimal overhead to the existing instruction set and hardware organization, and as a result it has been used in a number of recent microprocessors [79, 80, 81].

4.8 MIMD Architectures

In all of the architectures that we have discussed so far, all hardware resources were controlled by a single stream of instructions. In other words, the instruction control mechanism had a single control thread. MIMD (Multiple-Instruction, Multiple-Data) architectures allow the functional units to have their own independent control units. The

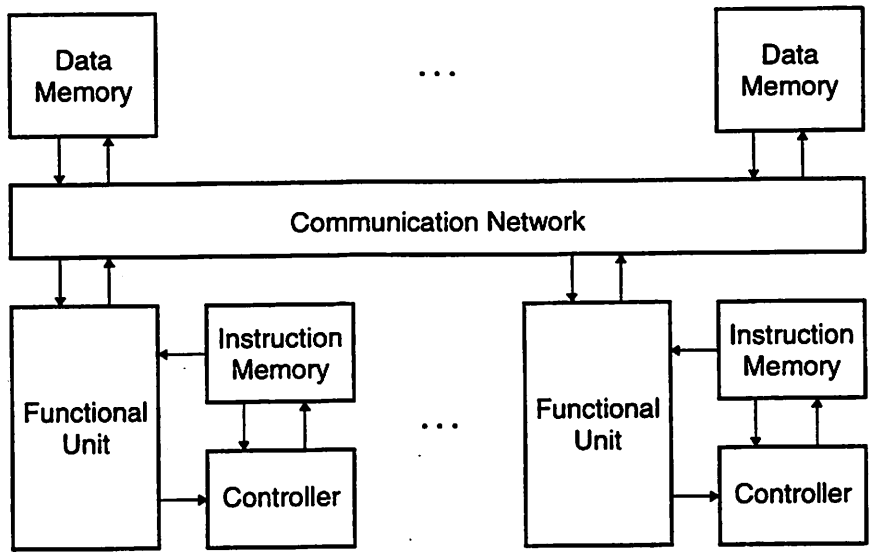


Figure 4.8: Basic Architectural Model for MIMD Processors

basic architectural model for MIMD processors is shown in Figure 4.8. Each functional unit is controlled by a local stream of instructions with a local controller. We can think of a MIMD processor as a processor with multiple SISD processors, and as a result MIMD architectures are also known as multiprocessors. Allowing each functional unit to have its own controller makes MIMD architectures highly flexible, and MIMD processors can generally be programmed to achieve very high performance for a wide variety of applications. In addition, the energy overhead of broadcasting instructions and control signals is avoided. However, there are now multiple controllers, so there is additional energy consumption that must be taken into consideration. The topology and energy overhead of the communication network is another design issue that requires careful attention. Multiprocessor DSPs have received a great deal of research interest and numerous architectures have been proposed and explored by researchers [82, 83, 84, 85].

One notable multiprocessor DSP architecture is the PADDI-2 architecture proposed by Yeung [84, 86], which was developed for rapid prototyping of video algorithms.

The PADDI-2 architecture is based on an array of 16-bit fine-grain nanoprocessors. Each nanoprocessor has its own local instruction memory that can store 8 instructions. The basic idea is to directly map the data flow diagram of a DSP algorithm onto the nanoprocessor array. The small local program at each nanoprocessor implements a node or a cluster of a few nodes of the data flow graph. The arcs of the data flow graph are implemented by a flexible interconnect network that can be configured by programming SRAM cells controlling switches in the interconnect network to create point-to-point links between the nanoprocessors. To avoid the overhead of a complete cross-bar network, while still providing a high degree of flexibility that can be used to create a wide variety of communication patterns, PADDI-2 uses a hierarchical two-level structure. The level-one network is used to create local connections between clusters of four nanoprocessors. The level-one networks can be connected to each other through a level-two network that allows nanoprocessors in different clusters to talk to each other. Computational activities are coordinated by a distributed data-driven control strategy in which nanoprocessor computations are synchronized by passing data and control tokens. Each nanoprocessor has input FIFOs that capture incoming tokens from the communication network. The strength of this distributed control mechanism is that it is highly scalable in supporting concurrent processing with a large number of nanoprocessors. The Pleiades architecture borrows from and builds on the lessons learned from the PADDI-2 architecture.

4.9 Field-Programmable Gate Arrays

Field-Programmable Gate Arrays (FPGA) were initially developed for prototyping and glue logic purposes, but advances in CMOS technology have allowed the development of high-capacity FPGAs that can be used to implement serious computing devices. The basic functional unit in an FPGA is a bit-processing element, which is commonly known as a Configurable Logic Block (CLB). The granularity of the functional units in an FPGA is thus at the finest possible level. An FPGA is a large array of CLBs. These CLBs

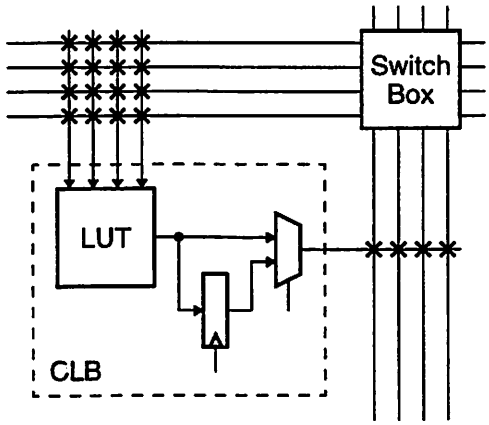


Figure 4.9: Basic CLB + Switch Matrix Tile of an FPGA

can be connected to each other in various desired ways by configuring a flexible interconnect network (see Figure 4.9). A CLB can implement any Boolean function of a small number, typically 4 to 5, of input bit operands. These functions are realized by using a look-up table (LUT) structure, i.e., a small SRAM memory, and the input bit operands serve as the address input of the LUT. Any Boolean function of the input operands can thus be realized by programming the LUT memory. One important innovation, pioneered by devices from Xilinx [87], was to allow the LUT memory to be used as a small, local random-access memory. Thus, a CLB can be used to implement both logic functionality and storage. The output of the LUT can optionally be registered if so desired by programming an SRAM cell the controls a multiplexer selecting either the output of the LUT or the registered version of the LUT output. The interconnection network consists of stretches of wires that can be connected to each other and to the CLBs by turning switches on and off. Each switch is controlled by an SRAM cell, that configures the switch to be on or off. An FPGA can be configured to implement any desired function by programming the SRAM cells that configure the LUTs and the interconnect switches. This feature makes FPGAs highly flexible and combined with the large number of CLBs available in modern CMOS processes, has resulted in tremendous interest in FPGAs as computing

devices. One big advantage of the FPGAs is that the functionality of hardware resources is decided after fabrication by the end user. In programmable processors, the functionality of the hardware resources is fixed after fabrication, and the end user is restricted to implementing the desired functionality by creating a sequence of instructions that tell these prefabricated hardware resources what to do. In FPGAs however, the user can directly implement the desired functionality by configuring just the right amount of hardware resources, i.e., CLBs and wires. As a result the computational throughput per unit silicon area for FPGAs can be much higher than programmable processors [88]. Computing machines based on FPGAs have been able to exceed the performance of supercomputers at a tiny fraction of the cost [89].

In FPGA devices, the instructions are the configuration bits stored in the SRAM cells controlling the LUTs and the switches. Once the FPGA is configured by loading the configuration SRAM cells with proper values, the functionality of the hardware resources are fixed. The instructions are distributed throughout the device, and directly control the LUTs and the switches. There is no energy overhead associated with fetching and decoding instructions. From this point of view, FPGAs are ideal. The one shortcoming of FPGAs, however, is that the configuration SRAM cells are typically programmed serially, which is a very slow process. Also, since the granularity of FPGAs is at the bit level, there is a tremendous amount of configuration information that must be loaded. These factors make reconfiguration a slow process.

FPGAs cannot however be considered as energy efficient devices. In fact, the opposite is true. The area and energy overhead of the interconnect network in FPGAs is substantial. 65% of the total energy in the Xilinx XC4003A FPGA is due to the wires, and another 21% and 9% are taken by clocks and I/O. The CLBs are responsible for only 5% of the total energy consumption [90]. When normalized to the same 0.6-μm process and

supply voltage used for the custom design shown in Figure 4.2, the energy consumed by the Xilinx XC4003A FPGA is 2.2 nJ per tap, and the maximum sample rate is 2.2 MHz.

The chief weakness of FPGAs is in the very fine granularity of the CLBs. This results in a great deal of overhead when implementing wide-word datapaths where there is no real need to control individual bits of the datapath independently.

4.10 Summary

Conventional programmable architectures are far less energy efficient than custom, application-specific devices. The cause of this inefficiency is the manner in which flexibility is achieved in conventional processors. Computations are performed on general-purpose functional units that are designed to implement a wide variety of arithmetic and logic functions. As a result, these functional units are large and complex, and their granularity is not always well-matched to the data types and the computations required by target algorithms. Data operands are stored in general-purpose memory units that are large, centralized structures. The tasks performed by these hardware resources during every execution cycle are specified by a stream of instructions that must be fetched from the instruction memory and then decoded and dispatched by the instruction controller. The net result is that a great deal of energy overhead is attached to every basic computational step. This basic weakness afflicted all of the architectures that we discussed in this chapter. In our quest to design highly energy efficient programmable architectures we should keep the following ideas in mind:

- One basic problem with conventional processors is that they are designed to be completely general-purpose. Architectures that target a smaller set of applications can be more efficient than general-purpose devices and must be pursued. While structurally similar to general-purpose processors, programmable signal processors are much more efficient because they are more customized for DSP algo-

rithms. Domain-specific architectures can be particularly efficient, as they provide the architect with the opportunity to match architectural parameters to the properties of the target domain of algorithms.

- Exploiting concurrency is the key to reducing energy consumption by reducing the supply voltage. Any energy efficient architecture must be able to support concurrent processing in an efficient and scalable manner. As we saw in Chapter 3, signal processing algorithms are highly amenable to concurrent implementations. This is a valuable opportunity that must be exploited.
- The overhead of instructions must be minimized. Vector processors and SIMD processors reduce the energy overhead of instructions by introducing vector instructions that can replace an entire program loop. This is an important technique that can be exploited for signal processing applications.
- FPGAs are ideal from the point of view of instructions because once an FPGA is configured, there is no overhead associated with fetching and decoding instructions. Reconfiguration of hardware resources is thus an important technique that can significantly reduce the overhead of instructions.
- The control structure used in a concurrent architecture is an important architectural issue that has a significant impact on scalability, efficiency, and ease of programming. Distributed control mechanisms, with multiple control threads, are better in this respect than centralized control schemes with a single thread of control.

The design of the Pleiades architecture template was heavily influenced by these considerations.

CHAPTER 5

Pleiades: Architecture Design

The Pleiades architecture will be presented in this chapter. We will first summarize the goals and the general architectural approach that motivated the design choices that were made. We will then present the Pleiades architecture template and explain its different components and their interactions. Architectural design of Maia, a domain-specific processor for CELP-based speech-coding that is based on the Pleiades architecture template will be presented next. We will demonstrate how algorithms are mapped onto a Pleiades-style processor using the Maia design.

5.1 Goals and General Approach

The approach that was taken in this work, given the overall goal of designing energy-efficient programmable architectures for digital signal processing applications, was to design processors that are optimized for a given domain of signal processing algorithms. This approach yields domain-specific processors, as opposed to general-purpose processors, which are completely flexible but highly inefficient, or application-specific processors, which are the most efficient but very inflexible. The intent is to develop a pro-

cessor that can, by virtue of its having been optimized for an algorithm domain, achieve high levels of energy efficiency, approaching that of an application-specific design, while maintaining a degree of flexibility such that it can be programmed to implement the variety of algorithms that belong to the domain of interest.

Algorithms within a given domain of signal processing algorithms, such as CELP-based speech coding algorithms, have in common a set of dominant kernels that are responsible for a large fraction of total execution time and energy. In a domain-specific processor, this fact can be exploited such that these dominant kernels are executed on highly optimized hardware resources that incur a minimum of energy overhead. This is precisely the approach that was taken in developing the Pleiades architecture.

An important architectural advantage that can be exploited in a domain-specific processor is the use of heterogeneous hardware resources. In a general-purpose processor, using a heterogeneous set of hardware resources cannot be justified because some of those resources will always be wasted when running algorithms that do not use them. For example, a fast hardware multiplier can be quite useful for some algorithms, but it is completely unnecessary for many other algorithms. Thus, general-purpose processors tend to use general-purpose hardware resources that can be put to good use for all types of different algorithms. In a domain-specific processor, however, using a heterogeneous set of hardware resources is a valid approach, and must in fact be emphasized. This approach allows the architect a great deal of freedom in matching important architectural parameters, particularly the granularity of the processing elements, to the properties of the algorithms in the domain of interest. Even within a given algorithm, depending on the particular set of computational steps that are required, there typically are different data types and different operations that are best supported by processing elements of varying granularity, and this capability can be provided by a domain-specific design. This is precisely one of the key

factors that makes an application-specific design so much more efficient than a general-purpose processor, where all operations are executed on processing elements with pre-determined architectural parameters that cannot possibly be a good fit to the various computational tasks that are encountered in a given algorithm.

Our overall objective of designing energy-efficient programmable processors for signal processing applications, and our approach of designing domain-specific processors, given the background of the preceding three chapters, can be distilled into the following architectural goals:

- Dominant kernels must be executed on optimized, domain-specific hardware resources that incur minimal control and instruction overhead. The intent is to increase energy efficiency by creating a good match between architectural parameters and algorithmic properties.
- Reconfiguration of hardware resources will be used to achieve flexibility while minimizing the energy overhead of instructions. As we saw in Chapter 4, FPGAs do not suffer from the overhead of fetching and decoding instructions. However, the ultra-fine granularity of the bit-processing elements used in FPGAs incurs a great deal of overhead for word-level arithmetic operations and needs to be addressed.
- To minimize energy consumption, the supply voltage must be reduced aggressively. To compensate for the performance loss associated with reducing the supply voltage, concurrent execution must be supported. The relative abundance of concurrency in DSP algorithms provides a good opportunity to accomplish this objective.
- The ability to use different optimal voltages for different circuit blocks is an important technique for reducing energy consumption and must be supported. This

requires that the electrical interfaces between circuit modules be independent of the varying supply voltages used for different circuit modules.

- Dynamic scaling of the supply voltage is an important technique to minimize the supply voltage, and hence energy consumption, to the absolute minimum needed at any given time and must be supported.
- The structure of the communication network between the processing modules must be flexible such that it can be reconfigured to create the communication patterns required by the target algorithms. Furthermore, to reduce the overhead of this network, hierarchy and reduced voltage swings will be used. The electrical interface used in the communication network must not be a function of the supply voltages of the modules communicating through the network.
- In order to avoid the large energy overhead of accessing large, centralized hardware resources, e.g. memories, datapaths, and buses, locality of reference must be preserved. The ability to support distributed, concurrent execution of computational steps is the key to achieving this goal, and it is also consistent with our goal of highly concurrent processing for the purpose of reducing the supply voltage.
- A key architectural issue in supporting highly concurrent processing is the control structure that is used to coordinate computational activities among multiple concurrent hardware resources. The control structure has a profound effect on how well an architecture can be scaled to match the computational characteristics of the target algorithm domain. The performance and energy overheads of a centralized control scheme can be avoided by using a distributed control mechanism. Ease of programming and high-quality automatic code generation are also important issues that are influenced by the control structure of a programmable architecture.
- Unnecessary switching activity must be completely avoided. There must be zero switching activity in all unused circuit modules.

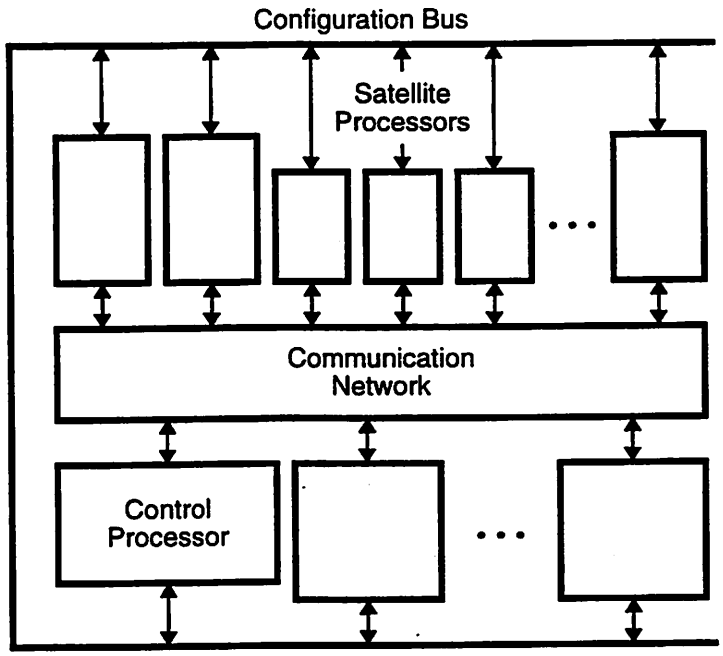


Figure 5.1: The Pleiades Architecture Template

- Time-sharing of hardware resources must be avoided, so that temporal correlations are preserved. This objective is consistent with and is in fact satisfied by our approach of relying on spatial and concurrent processing. Point-to-point links in the communication network, as opposed to time-shared bus connections, should be used to transmit individual streams of temporally-correlated data streams.

5.2 The Pleiades Architecture Template

In this section, a general overview of the Pleiades architecture will be presented. Additional details and architectural design issues will be presented and discussed in the following sections. Architectural design of Maia, a Pleiades-style processor for CELP-based speech coding algorithms will be presented subsequently.

The Pleiades architecture is based on the template shown in Figure 5.1. This template is reusable and can be used to create an instance of a domain-specific processor,

which can then be programmed to implement a variety of algorithms within the given domain of interest. All instances of this architecture template share a fixed set of control and communication primitives. The type and number of processing elements in a given domain-specific instance, however, can vary and depend on the properties of the particular domain of interest.

The architecture template consists of a *control processor*, a general-purpose microprocessor core, surrounded by a heterogeneous array of autonomous, special-purpose *satellite processors*. All processors in the system communicate over a reconfigurable communication network that can be configured to create the required communication patterns. All computation and communication activities are coordinated via a distributed data-driven control mechanism. The dominant, energy-intensive computational kernels of a given DSP algorithm are implemented on the satellite processors as a set of independent, concurrent threads of computation. The rest of the algorithm, which is not compute-intensive, is executed on the control processor. The computational demand on the control processor is minimal, as its main task is to configure the satellite processors and the communication network (via the configuration bus), to execute the non-intensive parts of a given algorithm, and to manage the overall control flow of the algorithm.

In the model of computation used in the Pleiades architecture template, a given application implemented on a domain-specific processor consists of a set of concurrent communicating processes [91] that run on the various hardware resources of the processor and are managed by the control processor. Some of these processes correspond to the dominant kernels of the given application program and run on satellite processors under the supervision of the control processor. Other processes run on the control processor under the supervision of a simple interrupt-driven foreground/background system for relatively simple applications or under the supervision of a real-time kernel for more complex

applications [92]. The control processor configures the available satellite processors and the communication network at run-time to construct the dataflow graph corresponding to a given computational kernel directly in hardware. In the hardware structure thus created, the satellite processors correspond to the nodes of the dataflow graph, and the links through the communication network correspond to the arcs of the dataflow graph. Each arc in the dataflow graph is assigned a dedicated link through the communication network. This ensures that all temporal correlations in a given stream of data are preserved and the amount of switching activity is thus minimized.

As we saw in Chapter 2, algorithms within a given domain of applications, e.g., CELP-based speech coding, share a common set of operations, e.g., LPC analysis, synthesis filtering, and codebook search. When and how these operations are performed depend on the particular details of the algorithm being implemented and are managed by the control processor. The underlying details and the basic parameters of the various computational kernels in a given domain vary from algorithm to algorithm and are accommodated at run-time by the reconfigurability of the satellite processors and the communication network.

The Pleiades architecture enjoys the benefit of reusability because (a) there is a set of predefined control and communication primitives that are fixed across all domain-specific instances of the template, and (b) predefined satellite processors can be placed in a library and reused in the design of different types of processors.

5.3 The Control Processor

A given algorithm can be implemented in its entirety on the control processor, without using any of the satellite processors. The resulting implementation, however, will be very inefficient: it will be too slow, and it will consume too much energy. To achieve good performance and energy efficiency, the dominant kernels of the algorithm must be

identified and implemented on the satellite processors, which have been optimized to implement those kernels with a minimum of energy overhead. Other parts of the algorithm, which are not compute-intensive and tend to be control-oriented, can be implemented on the control processor. The computational load on the control processor is thus relatively light, as the bulk of the computational work is done by the satellite processors.

In addition to executing the non-compute-intensive and control-oriented sections of a given algorithm, the control processor is responsible for *spawning* the dominant kernels as independent threads of computation, running on the satellite processors. In this capacity, the control processor must first configure the satellite processors and the communication network such that a suitable hardware structure for executing a given kernel is created. The satellite processors and the communication network are reconfigured at run-time, so that different kernels are executed at different times on the same underlying reconfigurable hardware fabric. The functionality of each hardware resource, be it a satellite processor or a switch in the communication network, is specified by the *configuration state* of that resource, a collection of bits that instruct the hardware resource what to do. The configuration state of each hardware resource is stored locally in a suitable storage element, i.e., a register, a register file, or a memory. Thus, storage for the configuration states of the hardware resources of a processor are distributed throughout the system. These configuration states are in the memory map of the control processor and are accessed by the control processor through the reconfiguration bus, which is an extension of the address/data/control bus of the control processor.

Once the satellite processors and the communication network have been properly configured, the control processor must initiate the execution of the kernel at hand. This is accomplished by generating a request signal to an appropriate satellite processor which will trigger the sequence of events whereby the kernel is executed. After initiating the exe-

cution of the kernel, the control processor can either halt (to save power) and wait for the completion of the kernel, or it can start executing another computational task, including spawning another kernel on another set of satellite processors. This mode of operation allows the programmer to increase processing throughput by taking advantage of coarse-grain parallelism. When the execution of the kernel is completed, the control processor receives an interrupt signal from the appropriate satellite processor. The interrupt service routine will determine the next course of action to be taken by the control processor.

5.4 Satellite Processors

The computational core of the Pleiades architecture consists of a heterogeneous array of autonomous, special-purpose satellite processors. These processors are optimized to execute specific tasks efficiently and with minimal energy overhead. Instead of executing all computations on a general-purpose datapath, as is commonly done in conventional programmable processors, the energy-intensive kernels of an algorithm are executed on optimized datapaths, without the overhead of fetching and decoding an instruction for every single computational step.

Kernels are executed on satellite processors in a highly concurrent manner. A cluster of interconnected satellite processors that implements a kernel processes data tokens in a pipelined manner, as each satellite processor forms a pipeline stage. In addition, each satellite processor can be further pipelined internally. Furthermore, multiple pipelines corresponding to multiple independent kernels can be executed in parallel. These capabilities allow efficient processing at very low supply voltages. For bursty applications with dynamically varying throughput requirements, dynamic scaling of the supply voltage is used to meet the throughput requirements of the algorithm at the minimum supply voltage.

As mentioned earlier, satellite processors are designed to perform specific tasks. Let us consider some examples of satellite processors:

- Memories are ubiquitous satellite processors and are used to store the data structures processed by the computational kernels of a given algorithm domain. The type, size, and number of memories used in a domain-specific processor depend on the nature of the algorithms in the domain of interest.
- Address generators are also common satellite processors that are used to generate the address sequences needed to access the data structures stored in memories in the particular manner required by the kernels.
- Reconfigurable datapaths can be configured to implement the various arithmetic operations required by the kernels.
- Programmable gate array (PGA) modules can be configured to implement various logic functions, as needed by the computational kernels.
- Multiply-Accumulate (MAC) processors can be used to compute vector dot products very efficiently. MAC processors can be useful in a large class of important signal processing algorithms.
- Add-Compare-Select (ACS) processors can be used to implement the Viterbi algorithm efficiently. The Viterbi algorithm is widely used in many communication and storage applications.
- Discrete Cosine Transform (DCT) processors can be used to implement many image and video compression/decompression algorithms efficiently.

Observe that while most satellite processors are dedicated to performing specific tasks, some satellite processors might support a higher degree of flexibility to allow the implementation of a wider range of kernels. The proper choice of the satellite processors used in a given domain-specific processor depends on the properties of the domain of interest and must be made by careful analysis of the algorithms belonging to that domain.

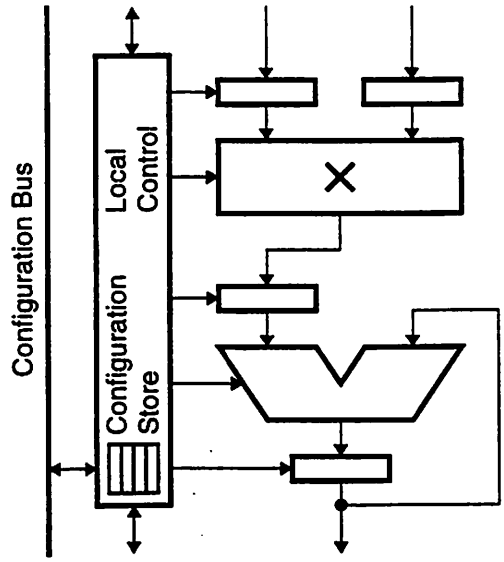
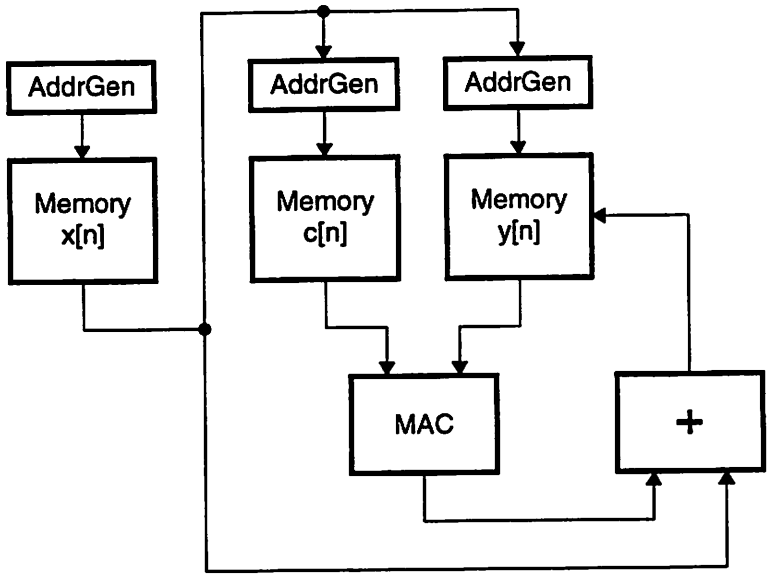


Figure 5.2: Block Diagram of a MAC Satellite Processor

The behavior of a satellite processor is dictated by the configuration state of the processor. The configuration state of a satellite processor is stored in a local configuration store and is accessed by the control processor via the reconfiguration bus. For some satellite processors, the configuration state consists of a few basic parameters that determine what the satellite processor will do. For other satellite processors, the configuration state may consist of sequences of basic instructions that are executed by the satellite processor. Instruction sets and program memories for the latter type of satellite processors are typically shallow, as satellite processors are typically designed to perform a few basic operations, as required by the kernels, very efficiently. As such, the satellite processors can be considered *weakly* programmable. For a memory satellite processor, the contents of the memory make up the configuration state of the processor.

Figure 5.2 shows the block diagram of a MAC satellite processor. Figure 5.3 illustrates how one of the energy-intensive functions of the VSELP speech coder, the weighted synthesis filter, is mapped onto a set of satellite processors.



$$y[1 \le n \le N-1] = x[n] + \sum_{i=1}^{N_P} c_i \cdot y[n-i]$$

Figure 5.3: The VSELP Synthesis Filter Mapped onto Satellite Processors

5.5 Communication Network

In the Pleiades architecture, the communication network is configured by the control processor to implement the arcs of the dataflow graph of the kernel being implemented on the satellite processors. As mentioned earlier, each arc in the dataflow graph is assigned a dedicated channel through the communication network. This ensures that all temporal correlations in a given stream of data are preserved, and the amount of switching activity is reduced.

The communication network must be flexible enough to support the interconnection patterns required by the kernels implemented on a given domain-specific processor, while minimizing the energy and area cost of the network. In principle, it is straightfor-

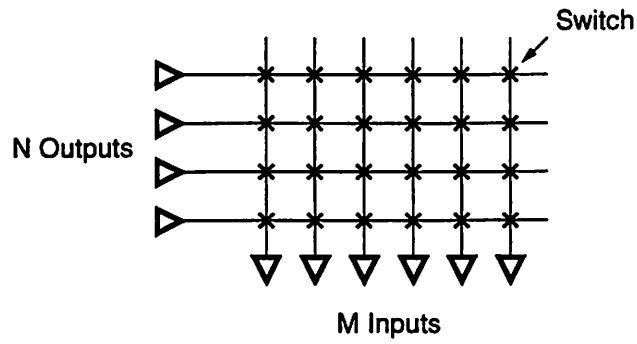


Figure 5.4: Crossbar Interconnection Network

ward to provide the flexibility needed to support all possible interconnection patterns for a given set of processors. This can be accomplished by a *crossbar* network, as shown in Figure 5.4. A crossbar network can support simultaneous, non-blocking connection of any of M input ports to any of N output ports. This can be accomplished by N buses, one per output port, and a matrix of $N \times M$ switches. The switches can be configured to allow a given input port to be connected to any of the output buses. However, the global nature of the buses and the large number of switches make the crossbar network prohibitively expensive in terms of both energy and area, particularly as the number of input and output ports increases. Each data transfer incurs a great deal of energy overhead, as it must traverse a long global bus loaded by N switches.

The number of switches can be reduced by using multi-stage interconnection networks [93], such as the *Omega* network [94] shown in Figure 5.5, which has been commonly used in many multiprocessor systems. As in the crossbar network, the number of buses for an $N \times N$ Omega network is N, but the number of switches is $N \log_2 N$, as an Omega network with N output ports and N input ports consists of $\log_2 N$ stages with $N/2$ switches per stage. Observe, however, that the switches of a multi-stage network are more complex than those of a crossbar network, as they must support more complex switching

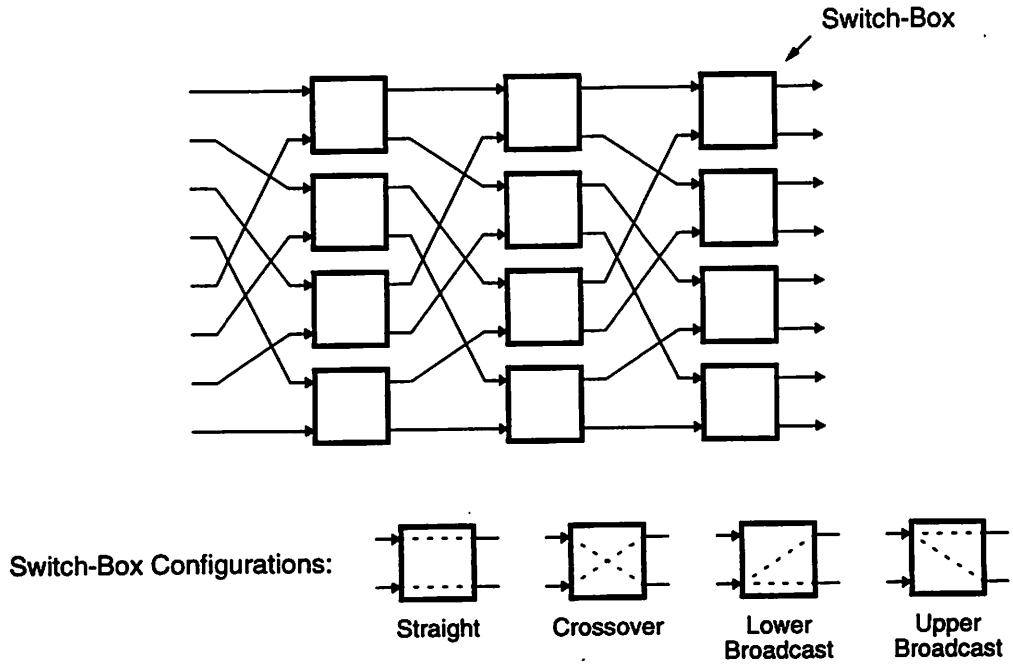


Figure 5.5: 8×8 Omega Multistage Interconnection Network

patterns, as illustrated in Figure 5.5. The complex routing patterns of multi-stage interconnection networks, such as the *perfect-shuffle* pattern used in the Omega network, make these networks particularly difficult and cumbersome to implement. Another drawback of multistage networks is that each connection through the network must go through $\log_2 N$ switches. This reduces the maximum data rate through each communication channel through the network.

In practice, a full crossbar network can be quite unnecessary and can be avoided. One reason is that not all output ports might be actively used simultaneously. Some output ports might in fact be mutually exclusive of one another. Therefore, the number of buses needed can be less than the number of output ports in the system. Another practical fact that can be exploited to reduce the complexity of a full crossbar (and other types of networks, as well) is that not all input ports need to be connected to all available output ports in the system. For example, address generators typically communicate with memories

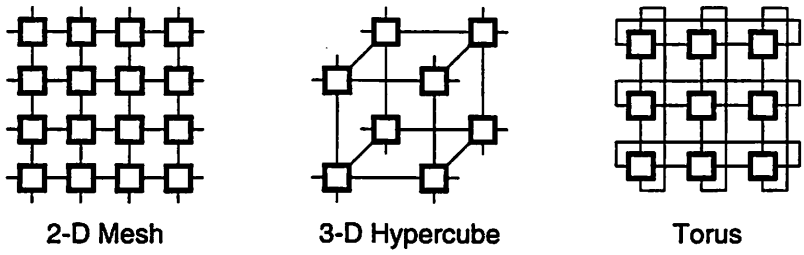


Figure 5.6: Some Examples of Network Topologies

only, and there is no need to allow for the possibility of connecting the address inputs of memory modules to the output ports of the arithmetic units. This fact can be used to reduce the span of the buses and the number of switches in the network. These techniques are employed in the Pleiades architecture.

The chief difficulty with the interconnect architectures discussed so far is the global nature of the buses. This makes all data transfers expensive regardless of whether they are between two adjacent processors or between two processors at opposite corners of the chip. The efficiency of data transfers can be improved by taking advantage of the fact that most data transfers are local. This is a direct manifestation of the principle of locality of reference discussed in Chapter 2. Instead of using buses that span the entire system, shorter bus segments are used that allow efficient local communication. Many such architectures have been proposed, particularly for use in multiprocessor systems, and some of them have been illustrated in Figure 5.6. These topologies provide efficient point-to-point local channels at the expense of long-distance communications. One simple scheme for transferring data between non-adjacent nodes is to route data tokens through other intervening processors. This increases the latency of data transfers, but keeps the interconnect structure simple. An additional drawback is that the latency of a data transfer becomes a function of processor placement and operation assignment. As a result, scheduling and assignment of operations become more complicated, and developing an efficient compiler becomes more difficult.

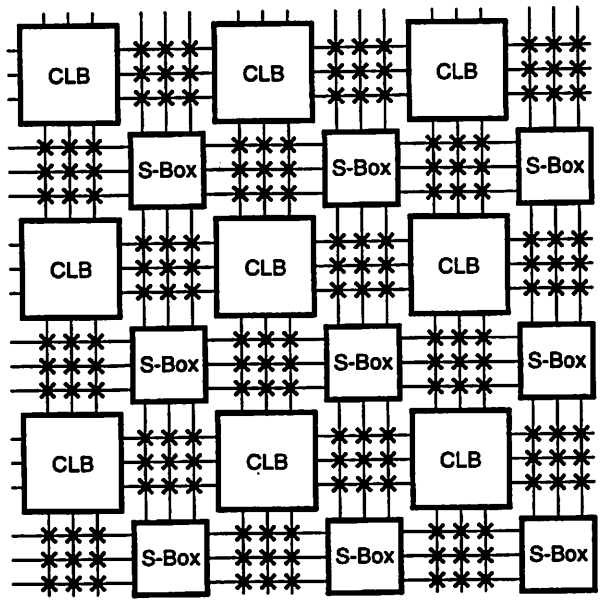


Figure 5.7: Simple FPGA Mesh Interconnect Structure

The mesh topology has been particularly popular in modern FPGAs. The mesh structure is simple and very efficient for VLSI implementations. A simplified version of the mesh structure, as used in many modern FPGAs, is illustrated in Figure 5.7. To transfer data between non-adjacent processing elements, multiple unit-length bus segments can be concatenated by properly configuring the switch-boxes that are placed at the boundaries of the processing elements. Local communications can be accomplished efficiently, and non-local communications can be supported, as well, and the degradation of communication bandwidth with distance, due to the increasing number of switches as more switch-boxes are traversed, is relatively graceful. This scheme has worked quite well in FPGAs, but it is not directly applicable to a Pleiades-style processor because a Pleiades-style processor is composed of a heterogeneous set of satellite processors with different shapes and sizes and the regular two-dimensional array structure seen in FPGAs cannot be created.

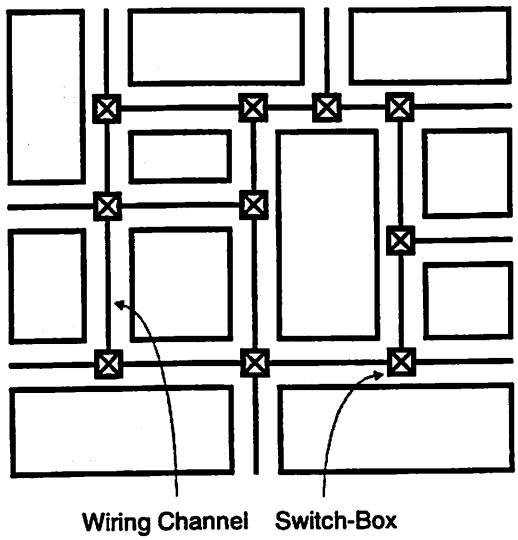


Figure 5.8: Generalized Mesh Interconnect Structure

The scheme used in the Pleiades architecture is a generalization of the mesh structure, i.e., a *generalized mesh* [95], which is illustrated in Figure 5.8. For a given placement of satellite processors, wiring channels are created along the sides of the satellite processors. Configurable switch-boxes are placed at the junctions between the wiring channels, and the required communication patterns are created by configuring these switch-boxes. The parameters of this generalized mesh structures are the number of buses employed in a given wiring channel, and the exact functionality of the switch-boxes. These parameters depend on the placement of the satellite processors and the required communication patterns among the satellite processors.

An important and powerful technique that can be used in improving the performance and efficiency of the communication network is the use of hierarchy. By introducing hierarchy, locality of reference can be further exploited in order to reduce the cost of long-distance communications. One approach that has been used in some FPGAs, e.g., the Xilinx XC4000 family [87], is to use a hierarchy of lengths in the bus segments used to

connect the logic blocks. Instead of using only unit-length segments, longer segments spanning two, four, or more logic blocks are also used. Distant logic blocks can be connected via these longer segments by using far less series switches than would have been needed if only unit-length bus segments were available.

Another approach to introducing hierarchy in the communication network is to use additional levels of interconnect that can be used to create connections among *clusters* of processing elements. An example of this approach is the two-level network structure used in the PADDI-2 multiprocessor [86], which was discussed in Chapter 4. In PADDI-2, a level-1 reduced crossbar network is used to connect nanoprocessors within clusters of four nanoprocessors. A level-2 reduced and segmented crossbar is used to create connections between the clusters. Another example of the application of hierarchy is the binary tree structure used in the Hierarchical Synchronous Reconfigurable Array architecture [96]. In this approach, a binary-tree hierarchy of switch-boxes is used to reduce the cost of communications between distant logic blocks. Local short-cuts are also used to facilitate efficient neighbor-to-neighbor connections, without the need to traverse the tree of switch-boxes.

In the Pleiades architecture, hierarchy is introduced into the communication network by creating clusters of tightly-connected satellite processors that internally use a generalized-mesh structure. Communication among clusters is accomplished by introducing inter-cluster switch-boxes that allow inter-cluster communication through the next higher level of the communication network. This is illustrated in Figure 5.9. The key challenge is the proper clustering of the satellite processors and the proper placement of the inter-cluster switch-boxes in order to avoid routing congestions. The proper organization can be found by closely studying the interconnection patterns that occur in the computational kernels of a given domain of algorithms.

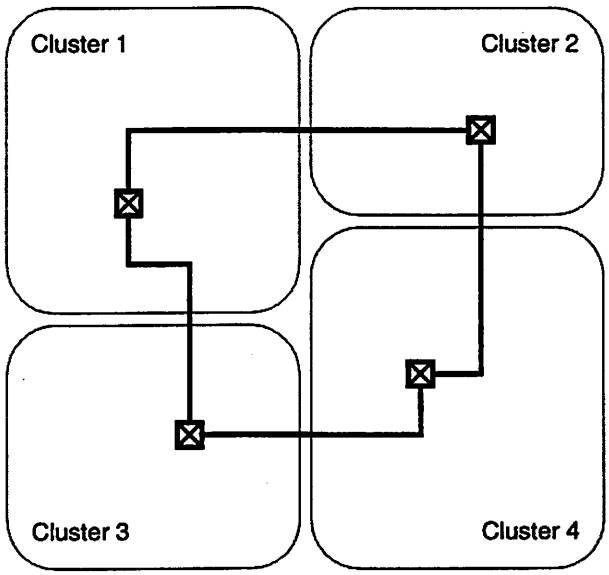


Figure 5.9: Hierarchical Generalized Mesh Interconnect Structure

In addition to the techniques mentioned above, the Pleiades architecture uses reduced-swing bus driver and receiver circuits to reduce the energy of data transfers through the network [97, 98]. An additional benefit of this approach is that the electrical interface through the communication network is standardized and becomes independent of the supply voltages of the communicating satellite processors. This facilitates the use of dynamic scaling of the supply voltage, as satellite processors at the two ends of a communication channel can run at independent supply voltages.

In summary, the Pleiades architecture uses a hierarchical generalized-mesh structure to provide the flexibility needed to implement the computational kernels of a given domain of algorithms on a heterogeneous set of satellite processors, while minimizing the energy overhead of data transfers among the satellite processors. Satellite processors communicate through point-to-point communication channels that are static for the duration of a kernel. Communication channels through the network correspond to the arcs of the data-flow graph of a given kernel. Communication links among the satellite processors are

established by the core processors by configuring the switches in the communication network. Use of point-to-point dedicated links ensures that temporal correlations are preserved, thus reducing switching activity. The communication network architecture used in the Pleiades architecture will be further evaluated within the context of the Maia processor later in this chapter.

5.6 Reconfiguration

In the Pleiades architecture, the flexibility needed to support the various kernels of a given domain of algorithms is achieved by the ability to reconfigure the satellite processors and the communication network at run-time, such that a hardware organization suitable for implementing a given kernel is created. This mode of programming is known as *spatial programming*, whereby the act of programming changes the physical interconnection of processing elements, thus creating a new hardware organization, i.e., a particular set of processing elements interconnected in a particular way, to implement a new computation. This is the mode of programming used in FPGAs. Traditional programmable processors rely on *temporal programming*, whereby the behavior of processing elements is altered in time, on a cycle-by-cycle basis, by a stream of instructions, and the underlying hardware organization is fixed.

As mentioned earlier, the behavior of satellite processors and the pattern of interconnections among them is dictated by the configuration state of the satellite processors and the switches in the communication network. Configuring a set of satellite processors or a set of switches in the communication network consists of altering the configuration state of these hardware resources by the control processor via the configuration bus. This is similar to what is done when programming FPGAs. However, in conventional FPGAs such as the Xilinx XC4000 family, reconfiguration is a very slow task that can take milliseconds of time. As a result, run-time reconfiguration is not practical with conventional

FPGAs. One basic reason for this shortcoming is that it takes a tremendous amount of configuration information to configure an FPGA. Part of the problem is the bit-level granularity of the processing elements. All details of the logic functions that are needed to implement a particular function must be fully specified. For example, it takes 360 bits of information to configure a Xilinx XC4000E CLB and its associated interconnect switches! The situation is further exacerbated when implementing word-level arithmetic operations, when a great deal of the configuration information is redundant and specifies the same logic functionality for different bits of a datapath. An additional obstacle to run-time reconfiguration is that FPGAs are typically configured in a bit-serial fashion[1]. The PADDI-2 DSP multiprocessor was also configured in a bit-serial manner, and as a result run-time reconfiguration was not practical, but this was not really a limitation for the design, as PADDI-2 was designed for rapid prototyping applications.

In the Pleiades architecture, since hardware resources are configured at run-time, so that different kernels can be executed on the same basic set of satellite processors at different times during the execution of an algorithm, a key design objective is to minimize the amount of time spent on configuring and re-configuring hardware resources. This can be accomplished with a combination of architectural strategies. The first strategy is to reduce the amount of configuration information. The word-level granularity of the satellite processors and the communication network is one contributing factor. No redundant configuration information is wasted on specifying the behavior of individual bit-slices of a multi-bit datapath. This is a direct result of the types of data tokens processed by signal processing algorithms. Another factor is that the behavior of most satellite processors (with the notable exception of PGA-style satellite processors) is specified by simple coarse-grain instructions choosing one of a few different possible operations supported by

1. In some recent devices, configuration information can be loaded into the device via a byte-wide bus [99].

a satellite processor and a few basic parameters, as necessary. For example, a MAC satellite processor can be fully configured by specifying whether to perform multiplication operations or to perform vector dot-product operations. Address generators can be configured by specifying one of a few different address sequences and specifying the associated address generation bounds, steps, and strides, as necessary. As a result, all it takes for the control processor to configure the satellite processors and the communication network is to load a few configuration store registers with the appropriate values.

Another strategy to reduce reconfiguration time in the Pleiades architecture is that configuration information is loaded into the configuration store registers by the control processor through a wide configuration bus, an extension of the address/data/control bus of the control processor. For example, with a 32-bit control processor, such as the ARM9 microprocessor core [100], configuration information can be loaded into the configuration store registers of the satellite processors and the communication network at a rate of 32 bits per cycle.

Another technique to minimize or even eliminate configuration time is to overlap configuration and kernel execution. While satellite processors are busy executing a kernel, they can be configured by the control processor for the next kernel to be executed. When the execution of the current kernel is completed, the satellite processors can start the next kernel immediately by switching to the new configuration state. This can be accomplished by allowing multiple configuration *contexts*, i.e., multiple sets of configuration store registers. This technique is similar to those used in multi-context and time-multiplexed FPGA devices [101, 102, 103]. While one configuration context is active and is used by the satellite processors and the communication network to execute the current kernel, a second passive configuration context is simultaneously loaded by the control processor in preparation for the next kernel. When the execution of the kernel is finished, the new context

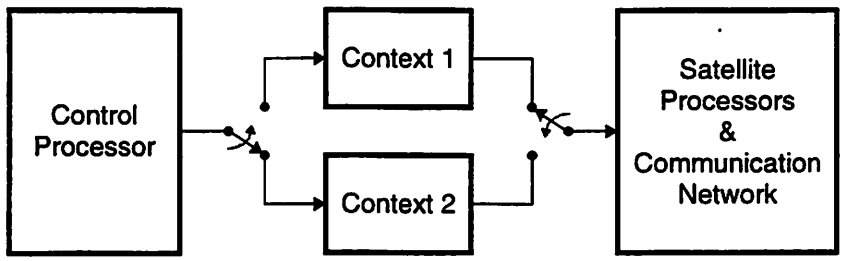


Figure 5.10: Concurrent Reconfiguration and Kernel Execution

becomes the active context, and the old context can be loaded with new configuration state in anticipation of the next kernel to be executed. This mode of operation is illustrated in Figure 5.10. An extension of this technique is to allow more than two configuration contexts, at least for some of the satellite processors. These configuration contexts can be preloaded when the system is initialized, and there will be no need to reconfigure the associated satellite processor at run-time. This latter technique was used in the address generators of the Maia processor.

5.7 Distributed Data-Driven Control

Coordination of computation and communication activities among the processing elements of a multiprocessor system is one of the most important architectural design issues, as it has a profound effect on the efficiency of the overall design. This task is performed by a suitable control mechanism. The responsibility of the control mechanism is to provide instructions to the processing elements, i.e., the functional units, the data memories, and the communication network (see Figure 4.1). In doing so, the control mechanism requires control information from the processing elements, indicating their current states to the control mechanism. How instructions are stored, how they are dispatched to the processing elements, and how control information provided by the processing elements is handled are the key issues that must be addressed in designing a control mechanism.

In the Pleiades architecture, computational kernels are executed on the satellite processors in a distributed, concurrent manner. This approach avoids the energy and performance overheads of large, centralized functional units and data memories by replacing global interactions across long distances by more local interactions across shorter distances. This same approach can be applied to the design of the control mechanism. The Pleiades architecture uses a *distributed* control mechanism that employs small local controllers in place of a large global controller.

In a *centralized* control mechanism, a single global controller is responsible for controlling the activities of all processing elements. VLIW and SIMD architectures, for example, use a centralized control mechanism. The conceptual simplicity of this scheme works well when there is a single thread of computation. In a multiprocessor system with multiple processors executing multiple threads of computation, however, a centralized control mechanism loses its conceptual simplicity and becomes quite cumbersome, as the controller has to deal with the combinatorial explosion of control states as the combined states of the individual processing elements are considered together. As a result, developing programs and compilers for architectures that use a centralized control mechanism becomes very complex and difficult. Furthermore, a centralized control mechanism incurs a great deal of energy and performance overhead because instructions to the processing elements and control information from the processing elements are all communicated globally through the central controller. As a result, a centralized control mechanism cannot practically be scaled up to deal with a large number of processing elements because the required bandwidth for distributing instructions and control information and the associated energy overhead and performance penalty can become prohibitive.

In a distributed control mechanism, each processing element has a local controller with a local program memory. As a result, the energy and performance overheads of stor-

ing and distributing instructions and communicating control information are greatly reduced as these interactions assume a local nature. With a distributed control mechanism, a computational problem can be partitioned into multiple threads of computation in the most natural way dictated by the problem itself, without the artificial constraints of a centralized control mechanism, and these threads of computation can then be distributed across multiple processing elements or multiple clusters of processing elements. The ability to take such a modular approach eases programming and developing compilers for an architecture with a distributed control mechanism. Another important advantage of a distributed control mechanism is that it can be gracefully scaled to handle multiprocessor systems with a large number of processing elements to tackle increasingly complex computational problems.

The key design issue with a distributed control mechanism is how a local controller coordinates its actions with other local controllers that it needs to interact with during the course of the execution of a given algorithm. One aspects of this problem is that each local controller must somehow determine *when* it can start executing a particular task. The objective here is to synchronize the actions of the controllers, so that computational activities are executed in the correct sequence. This can be accomplished by the exchange of *tokens* of control information among the controllers through the communication network, in the same way that data tokens are exchanged among the processing elements. Arriving control tokens can not only be used by a controller to determine when to initiate the next computational task, but depending on the control information encapsulated into the control tokens, they can also be used to determine which particular task is to be initiated by the controller.

Minimizing the overhead of control tokens is an important design issue in a distributed control mechanism. An even more fundamental issue is how to map a given algo-

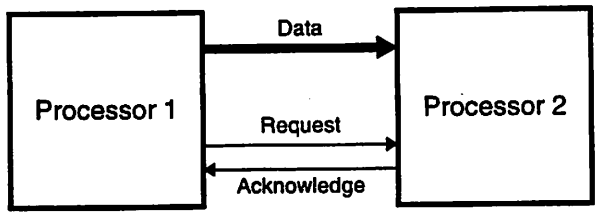


Figure 5.11: Data-Driven Execution via Handshaking

rithm onto processing elements that are controlled in a distributed manner. The approach taken in the Pleiades architecture is to map the dataflow graph of a given signal processing kernel directly onto a cluster of satellite processors interconnected through the communication network. In this approach, a satellite processor directly corresponds to a node or a cluster of nodes in the dataflow graph of a given kernel, and a communication channel through the communication network directly corresponds to an arc in the dataflow graph. Just as in the dataflow graph representation, the execution of an operation in a satellite processor is triggered by the arrival of all required data tokens, i.e., operations are executed in a *data-driven* manner [104]. Thus, data tokens not only provide the operands to be processed by the satellite processor, but they also implicitly provide synchronization information. A *handshaking* mechanism is required to implement a data-driven mode of operation: the arrival of a data token is signalled by a *request* signal from the sending satellite processor, and the acceptance of a data token is signalled by an *acknowledge* signal from the receiving satellite processor (see Figure 5.11). This approach to distributed control is similar to the control mechanism of the PADDI-2 architecture [86] and the DSP architecture proposed by Fellman [105]. As we will soon see, however, the particular control mechanism used in the Pleiades architecture provides additional support for handling common signal processing data structures such as vectors and matrices more efficiently.

The conceptual simplicity and elegance of data-driven distributed control greatly simplify the task of developing programs and compilers for the Pleiades architecture.

Extensive prior experience by researchers has demonstrated that dataflow graphs are perhaps the most natural and most effective means to represent signal processing algorithms [106, 107]. One of the key strengths of dataflow graphs is that they expose parallelism by expressing only the data dependencies that are inherent to a given algorithm. There is a rich body of knowledge addressing the problem of compiling dataflow graphs onto multiprocessor architectures [108, 109, 110, 111].

A data-driven control mechanism has another important benefit: it provides a well-defined and elegant framework for managing switching activity in hardware modules. The handshaking mechanism that is used to implement the data-driven semantics of dataflow graphs can also be used to control switching activity in the satellite processors. When all required data tokens have arrived at a satellite processor, the satellite processor can start executing its task; otherwise, the satellite processor will stay dormant, and no unnecessary switching activity will take place.

5.7.1 Control Mechanism for Handling Data Structures

Distributed execution of an algorithm on multiple processing elements involves partitioning the calculations performed by the algorithm into multiple threads. These threads are then assigned to appropriate processing elements. A convenient first step is to partition the algorithm into *address* calculations and *data* calculations. Address calculations produce memory address sequences that are used to access data structures in the particular manner specified by the algorithm. Data calculations process the accessed data structures and produce the desired results. This is illustrated in Figure 5.12 for the vector dot product example. Address calculations involve loop index and memory address pointer calculations. These calculations are mapped onto address generators. The address sequences produced by the address generators are used to access the required data structures (two vectors in this example) from the memory units. The resulting data streams are

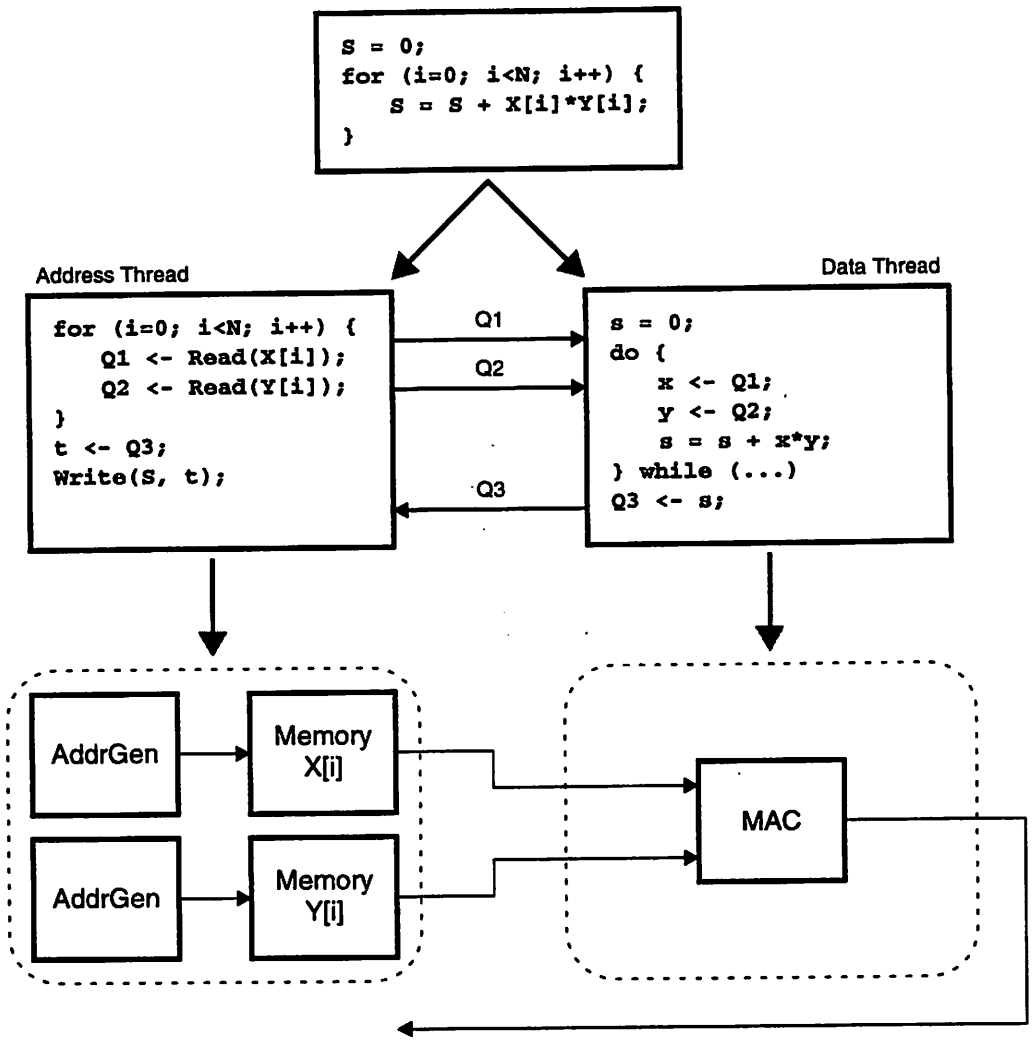


Figure 5.12: Address and Data Threads for Computing Vector Dot Product

then communicated to the functional units performing the data calculations (a single MAC unit in this example). The MAC unit must have a way of knowing when the end of a vector is reached. This information will provide the missing condition of the `while()` statement in the data thread in Figure 5.12. One approach is to replicate the loop index calculation of the address thread in the data thread. A better approach that avoids the overhead and inconvenience of replicating the loop index calculation is to let a data stream itself indicate the boundaries of the data structure that it is carrying. This can be done by

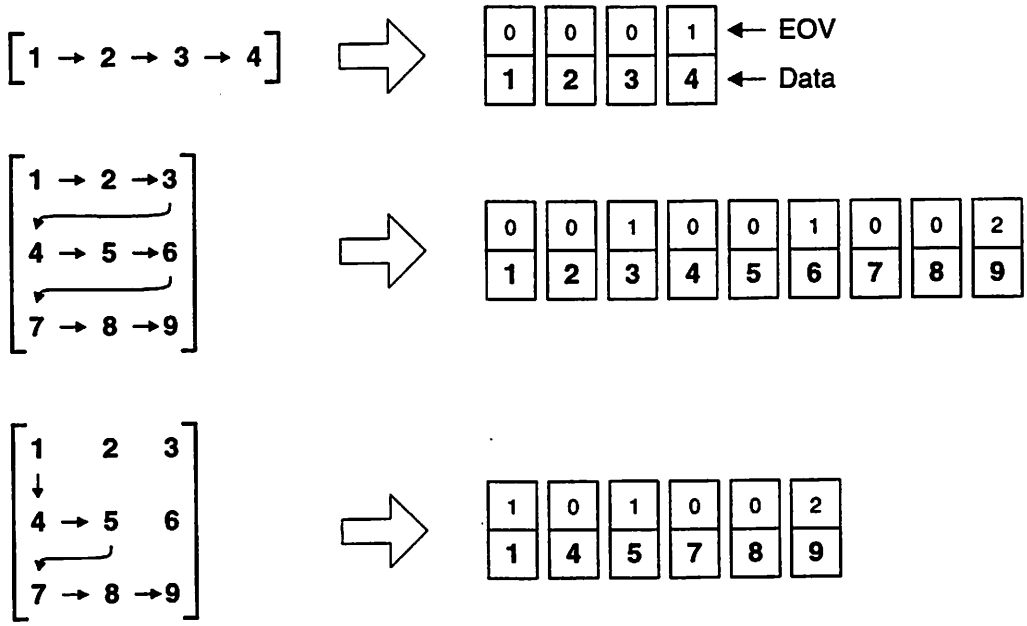
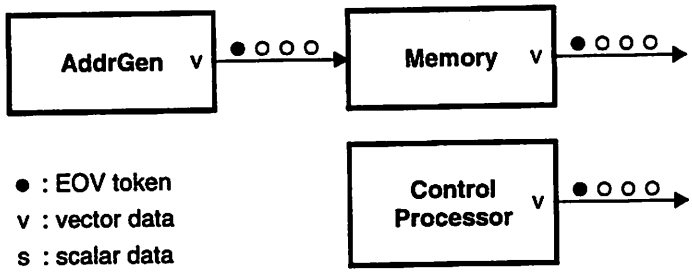


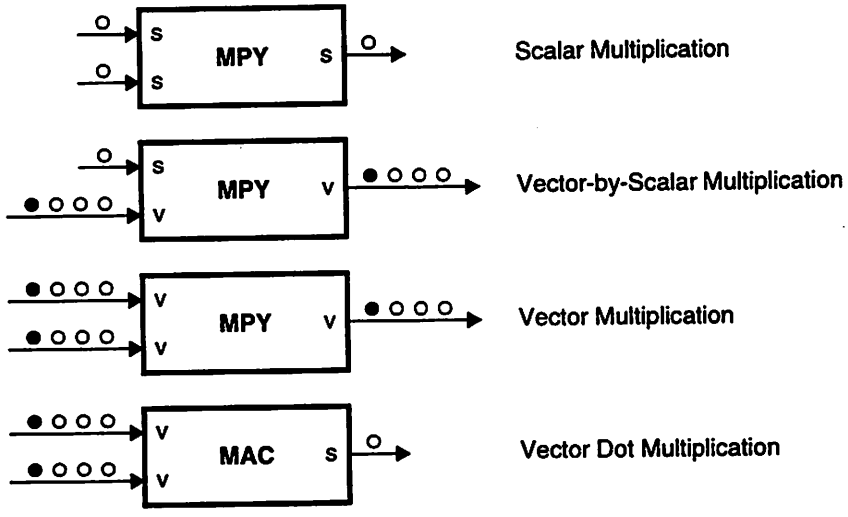
Figure 5.13: Data Stream Examples for Accessing Vectors and Matrices

embedding special control flags that indicate the last element of a sequence into data tokens. The latter approach was taken in the Pleiades architecture.

In the Pleiades architecture, a data stream can be a scalar, a vector, or a matrix. These data types are the most common in signal processing algorithms. The boundaries of vectors and matrices are indicated by special *end-of-vector* (EOV) flags that are embedded into data tokens. Figure 5.13 illustrates how this is accomplished. An EOV flag can have one of three values: 0, 1, or 2. The value 1 marks the last data token of a one-dimensional data structure or the last data token of a one-dimensional sub-structure of a two-dimensional data structure. The value 2 marks the last data token of a two-dimensional structure. The value 0 marks all other data tokens. Thus, two additional bits are needed to encode the EOV flag into a data token. Observe that the manner in which the elements of a vector or a matrix are scanned determines how the resulting data stream is delimited with EOV flags. Data structures of higher dimensions can also be created by allowing the EOV flag to take



(a) Data Stream Production



(b) Data Stream Consumption

Figure 5.14: Examples of Data Stream Production and Consumption

on more than three values. This was not deemed necessary for any of the Pleiades processors that were considered. EOV flags are inserted into data tokens by either an address generator producing the address sequence that is used to access the required data structure or by the control processor. Memory units simply copy the EOV flag of an incoming address token into the corresponding data token being read from memory. This is illustrated in Figure 5.14. How the EOV flags are used by a functional unit depends entirely on the instruction being executed by that functional unit, and the instruction being executed

by a functional unit must specify the type of and the manner in which an incoming data stream that is to be processed. Some examples of how data streams can be consumed by a satellite processor are shown in Figure 5.14 for the case of the MAC satellite processor. For any given instruction of a satellite processor, if the dimensionality of all input streams is increased by one, then the dimensionality of the output data stream is automatically increased by one, without the need to specify a new instruction. For instance, if the input data streams of the vector dot product instruction of the MAC processor are two-dimensional vectors instead of the one-dimensional vectors shown in Figure 5.14, then the output will automatically be a vector, with the proper EOV delimiters, instead of a scalar.

5.7.2 Summary

With its distributed data-driven control mechanism, the Pleiades architecture avoids the energy and performance overheads of communicating instructions and control signals globally across large distances, while providing modular and scalable support for highly concurrent implementations of signal processing algorithms. The control mechanism used in the Pleiades architecture provides support for handling common signal processing data structures such as vectors and matrices efficiently.

5.8 System Timing and Synchronization

Implementation of the handshaking mechanism that is needed in a data-driven control scheme is an important design issue. While the handshaking mechanism can be implemented within a conventional synchronous timing scheme with a global clock signal, where the status of the handshaking signals are examined on a cycle-by-cycle basis, as was done in the PADDI-2 design, data-driven control possesses an inherently asynchronous nature in which the arrival of data tokens at whatever point in time, not the tick of a global clock signal, is used to synchronize and coordinate computational steps. The handshaking mechanism can in fact be implemented, in a more natural way, within an asyn-

chronous timing scheme, in which the handshaking signals can also be used to synchronize data transfers to and from storage elements, without the need for a global clock signal [112].

In the Pleiades architecture, satellite processors communicate via an asynchronous timing scheme. This approach has a number of important benefits that are beyond the conceptual simplicity and elegance of combining data-driven control with asynchronous timing. The required throughput and the corresponding internal operating frequency of a satellite processor depend on the computational task that it is expected to perform and can vary from algorithm to algorithm or even from kernel to kernel. To minimize power dissipation, the internal operating frequency of any given satellite processor must be at the minimum required to meet the required processing throughput, and this is accomplished by setting the supply voltage of the satellite processor to the minimum required to meet the expected operating frequency. This can be done either statically or dynamically (by using dynamic scaling of the supply voltage). We thus have a situation in which multiple communicating processors can operate at different and time-varying internal operating frequencies and supply voltages. To accommodate multiple and time-varying operating frequencies, an asynchronous timing scheme is required, at least at the global level, for inter-satellite communication, because an asynchronous timing mechanism is independent of the operating frequencies of the communicating modules. One important benefit of this approach is that once a satellite processor has been designed and its functionality and internal timing have been verified and characterized, it can be utilized in a domain-specific processor without the need to re-verify its timing within the context of the overall processor because its internal timing is independent of external timing constraints. This is certainly not the case in a synchronous design, where the timing of a module is subject to external timing constraints, such as clock skew and the setup and hold times of the modules with which it must communicate. An asynchronous timing scheme results in a highly

modular design style in which a domain-specific processor can be constructed seamlessly by assembling the required set of satellite processors from a pre-designed library of processors, without having to re-design and re-verify existing satellite processors for a new set of external timing constraints.

An additional benefit of an asynchronous timing scheme is that the energy overhead of distributing a global clock signal is avoided. This can result in significant savings, particularly for high-performance designs, as the overhead of distributing a high-speed, low-skew clock signal can be quite high (as high as 40% of total power dissipation for some high-performance designs [113]). It is often cited [114], and it is certainly true, that with an asynchronous timing scheme, switching activity is minimized because the storage elements of an asynchronous circuit module are clocked and loaded with new values only when there is a request for a new computation; otherwise, the storage elements of that module are not clocked, and there is no switching activity in that module. In a straightforward synchronous implementation, the storage elements are always clocked even if there is no new data to be processed, and there is a great deal of unnecessary switching activity, wasting a great deal of energy. However, a synchronous system can be designed, using clock-gating techniques, such that its storage elements have the same switching profile as its asynchronous counterpart. Thus, a synchronous circuit module can be designed such that its storage elements are clocked and loaded with new data only when there is a new computation to be performed by that module. However, a global clock signal is still present and must be distributed to all circuit modules, and the required clock-gating control circuits incur additional overhead. The overhead of clock-gating control circuits is relatively minor and is comparable to the overhead of handshaking control circuits in an asynchronous design. Thus, the real advantage of an asynchronous design, from the point of view of energy, is that the overhead of the clock distribution network is avoided.

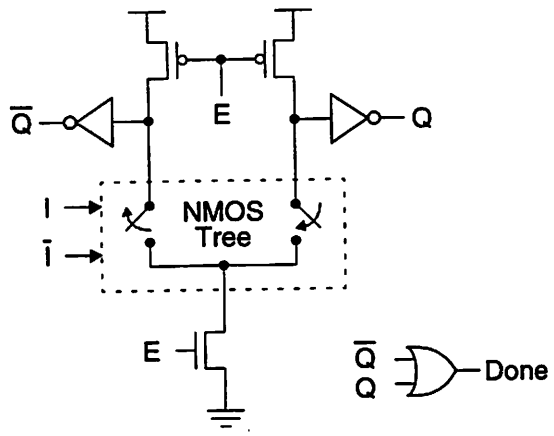


Figure 5.15: Completion Signal Generation in Asynchronous CMOS Circuits

An important issue in asynchronous systems is the overhead of generating completion signals. In a synchronous design, the availability of new data is indicated implicitly. The beginning of a new clock cycle, e.g., the rising edge of the clock signal, loads storage elements with new data to be processed by the combinational logic blocks, which are required to finish evaluating by the end of the current clock cycle, which is also the beginning of the next clock cycle, when their results will be loaded into the appropriate storage elements, and the next processing cycle will begin. In an asynchronous design, on the other hand, availability of new data is indicated explicitly. Each logic block generates a completion signal when it has finished evaluating. The completion signal results in a request signal to other blocks, informing them of the availability of new data to be processed by them. In asynchronous systems, completion signals can be generated by encoding each bit of data on a pair of signals. Figure 5.15 illustrates how this is accomplished in CMOS designs [115, 116]. A logic gate is implemented using a differential circuit. Initially, the gate is precharged, and both Q and $\overline{Q}$ outputs are high, indicating that the circuit is waiting to evaluate, i.e., it is not done yet. When all required inputs become available, and the gate evaluates, one of its outputs is discharged, indicating that it has finished eval-

uating. The completion signal (the Done signal in Figure 5.15) is the logical OR of Q and $\overline{Q}$. There are two serious problems with this approach. First, since a triplet of values (low, high, waiting) must be coded on a pair of signals, asynchronous logic gates, such as the differential CMOS gate shown in Figure 5.15, are more complex than the simple logic gates that are sufficient for synchronous designs. As a result, more capacitance is switched in each logic gate in an asynchronous circuit. Second, because of the precharge/evaluate and differential nature of asynchronous logic gates, switching activity is at maximum because for every evaluation, one side of the circuit must first be precharged, and the same or the other side of the circuit must be discharged. Thus, the requirement to generate completion signals incurs a heavy energy penalty that could outweigh the benefit of avoiding the overhead of clock distribution. This problem with asynchronous circuits motivated the use of synchronous techniques for the internal design of the satellite processors.

In the Pleiades architecture, each satellite processor consists of a *synchronous core* and an *asynchronous handshake controller*, as shown in Figure 5.16. The synchronous core has its own local clock signals. The core implements the basic functionality of the satellite processor. It can be a simple pipeline stage, a multi-stage pipeline, or a finite-state machine. The core communicates with other satellite processors through the handshake controller, which provides an asynchronous interface to the satellite processor. Satellite processors communicate using a common asynchronous handshake protocol, and there is no global clock signal. The handshake controller is responsible for synchronizing the transfer of data tokens to and from the core through the input and output ports of the satellite processor. The controller is also responsible for generating the clock signals needed by the synchronous core. The behavior of the handshake controller is determined by (a) the functionality of the satellite processor, (b) instructions from the configuration state of the

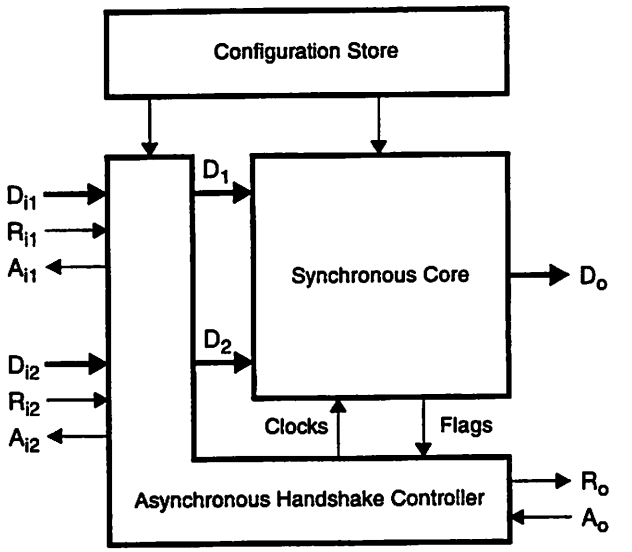


Figure 5.16: General Structure of a Satellite Processor

satellite processor, (c) control information from the synchronous core, and (d) control information embedded into the input data streams.

The handshake protocol that is used for inter-satellite data transfers is the two-phase protocol shown in Figure 5.17. The main reason for this choice was the higher performance of the two-phase protocol compared to the four-phase protocol, which is also shown in Figure 5.17. The two-phase protocol involves only two back-to-back transitions on the handshake signals, corresponding to one round-trip delay across the communication network, whereas the four-phase protocol requires four back-to-back transitions on the handshake signals (two round-trip delays). As a result, the maximum data rate with the two-phase protocol is higher than that of the four-phase protocol. The chief advantage of the four-phase protocol is its return-to-zero characteristic. Since typical latch and flip-flop circuits are activated by *levels* on the clock signal, as opposed to *transitions*, it is easy to generate the clock signals for the storage elements with the four-phase protocol. The four-

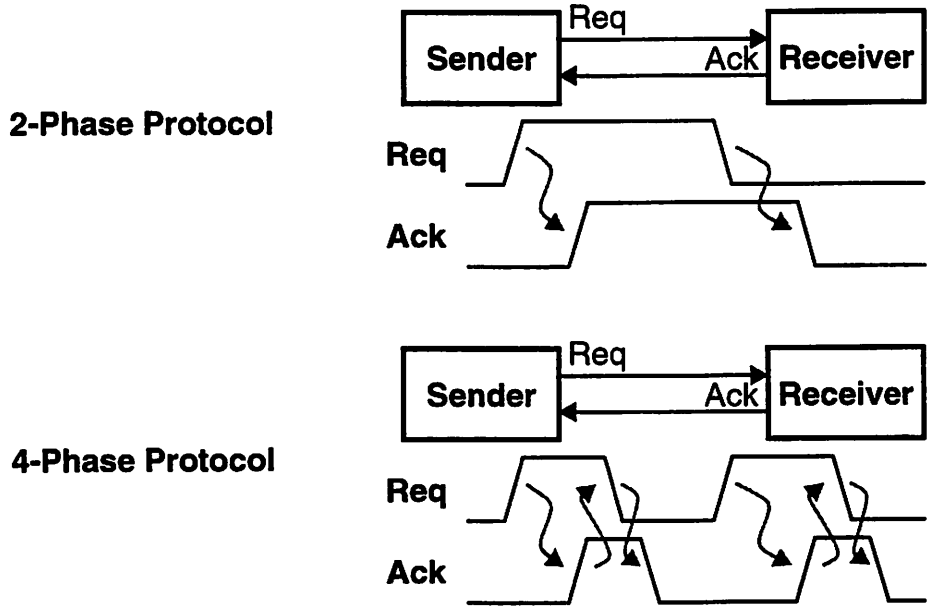


Figure 5.17: Asynchronous Handshake Protocols

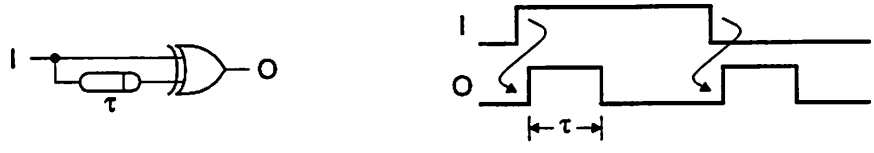


Figure 5.18: Transition-to-Pulse Converter

phase protocol is particularly convenient when used in conjunction with precharged dynamic logic [117, 118]. With the two-phase protocol, either the storage elements must respond to transitions, e.g., double-edge-triggered flip-flops, or the transitions must be converted to pulses. The latter approach was taken in the Pleiades implementations. Transitions can be converted to clock pulses with the circuit shown in Figure 5.18.

Since the handshake controller is responsible for generating local clock signals for the synchronous core, it must be able to estimate the cycle time of the core. This requires a timing reference that can model and track the most critical path in the core. One way to

accomplish this is to build a replica of the most critical path in the core. This approach is, however, not always practical or possible. Another approach is to have a timing chain built from delay elements with the proper delay and some safety margin. One approach is to use simple inverters as delay elements, but a more area-efficient and energy efficient approach is to use inverters built from long-channel transistors. With this approach, the power dissipation of the timing reference circuit can be kept to less than 1% of the total power dissipation of the satellite processor. One important issue with a timing reference circuit is how well it can track the cycle time of the core. Proper operation requires that a certain amount of safety margin be built into the delay through the timing reference circuit, but this margin will reduce the performance of the satellite processor if it is excessive. The approach taken in the Pleiades architecture was to use a programmable timing reference whose delay is set by the configuration state of the satellite processor. This approach allows the timing reference circuits to be configured during testing, so that the delay through the timing reference circuits is the minimum required for proper operation.

Figure 5.19 shows a simplified diagram of the handshake controller for a single-input/single-output satellite processor with a single pipeline stages. The design of the handshake controller circuits for the Maia processor was taken up by Martin Benes, and details of the design can be found in his Masters thesis [119].

An important point that should be mentioned is that the globally asynchronous, locally synchronous timing scheme that was chosen for the Pleiades architecture is strictly an implementation-related issue and is independent of the core architectural concepts of the Pleiades architecture. A Pleiades-style processor can also be implemented using a conventional synchronous timing scheme, but it will not benefit from the advantages of the timing scheme described above.

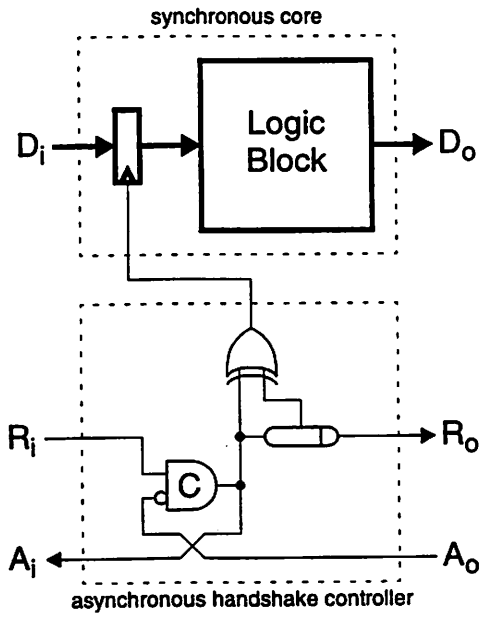


Figure 5.19: Example of a Handshake Controller

5.9 The Pleiades Design Methodology

The Pleiades approach is not only a hardware architecture for domain-specific processors, but it also involves an associated design methodology that is used to create domain-specific processor instances based on the Pleiades architecture template.

The Pleiades design methodology has two separate, but related, aspects that address different design tasks. One aspect of the methodology addresses the problem of designing a domain-specific processor for a given algorithm domain. The other aspect of the methodology addresses the problem of mapping a given algorithm onto an existing domain-specific processor instance. Both of these tasks involve analyzing algorithms and mapping them onto hardware resources. The chief difference between these two tasks is that in one of them, i.e., the problem of creating a domain-specific processor instance, architectural parameters (i.e., types and numbers of satellite processors and the detailed

structure of the communication network) are not fixed and are to be determined by the algorithm analysis and mapping process.

The design flow begins with a description of a given algorithm in C or C++. The baseline implementation is to map the entire algorithm onto the control processor. The power and performance of this baseline implementation are then evaluated and used as reference during subsequent optimizations, during which the objective will be to minimize energy consumption while meeting the real-time performance requirements of the given algorithm. The key task at this point is to identify the dominant kernels that are causing energy and performance bottlenecks. This is accomplished by dynamic profiling of the algorithm. Dynamic profiling establishes the function call graph of the algorithm and tabulates the amount of time and energy taken by each function and each basic block of the program. With this information, the dominant kernels of the algorithm can then be identified. The energy consumption of the baseline implementation is estimated using a modeling approach in which each instruction of the control processor has an associated base energy cost, and the total energy of a given program is obtained by adding the base costs of all executed instructions [120]. More accuracy can be obtained by taking account of inter-instruction energy consumption effects into the base costs of the instructions. A basic optimization step at this point, before going further into the rest of the design flow, is to improve the algorithm by applying architecture-independent optimizations and rewriting the initial description.

Once dominant kernels are identified, they are ranked in the order of importance and addressed one at a time until satisfactory results are obtained. One important step at this point is to rewrite the initial algorithm description, so that kernels that are candidates for being mapped onto satellite processors are distinct function calls. The next step is to implement a candidate kernel on an appropriate set of satellite processors. This is done by

```
1     int dot_product(int x[], int y[], int n)
2     {
3        int i;
4        int s;
5
6        s = 0;
7        for (i = 0; i < n; i++) s += x[i]*y[i];
8        return s;
9     }
```

Figure 5.20: C++ Description of Vector Dot Product

directly mapping the dataflow graph of the kernel onto a set of satellite processors. With this approach, each node or cluster of nodes in the dataflow graph corresponds to a satellite processor. Arcs of the dataflow graph correspond to links in the communication network, connecting the satellite processors. Mapped kernels are represented using an intermediate form as C++ functions that replace the original functions. The advantage of this approach is that mapped kernels can be simulated and evaluated with the rest of the program within the same environment that was used to simulate and evaluate the original program. In the intermediate form representation, satellite processors and communication channels are modeled as C++ objects. Each object has a set of methods that captures the functionality of the object during configuration and execution. This can be illustrated by an example. Figure 5.20 shows a C++ function implementing the vector dot product kernel. Figure 5.21 shows a mapping of the vector dot product kernel onto a set of satellite processors. Note that in this particular implementation of the vector dot product, both input vectors are stored in the same memory, and are communicated to the MAC satellite through the same communication channel in a time-multiplexed fashion. The MAC satellite is configured to accept both input vectors from the same input port (the other input port is unused). Figure 5.22 shows the intermediate form representation of the same func-

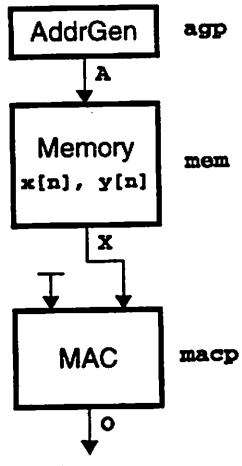


Figure 5.21: Mapping of Vector Dot Product

tion. The intermediate form representation is functionally identical to the original function but captures details of the actual implementation of the original function on satellite processors. In the intermediate form representation, first the required satellite processors and communication channels are instantiated. The satellite processors are then interconnected by configuring the communication channels. Finding the most efficient way to connect the required set of satellite processors through the communication network is a routing problem that is an important part of the overall design methodology [121]. The satellite processors are configured next. Configuration of the satellite processors and the communication network switches is performed by code running on the control processor. Automatic generation of this configuration code is an important part of the Pleiades design methodology [122]. The overhead of the configuration code must be minimized by scheduling the configuration code such that the amount of overlap between execution of the current kernel and configuration for the next kernel is maximized. The kernel is then executed. Notice that the execution of the kernel in this particular example is scheduled statically, but this is not a requirement, and by employing a thread library, the kernel can be executed as a set of concurrent processes, representing the concurrent hardware components. The energy and performance of the mapped kernels can then be estimated during simulation with macro-

```
1     int dot_product(int x[], int y[], int n)
2     {
3        Memory mem;
4        AGP agp;
5        MACP macp;
6        Queue A;    // output of agp, address input of mem
7        Queue X;    // data output of mem, X input of macp
8        Queue O;    // output of macp
9        Queue unused;    // dummy Queue for unused ports
10       int x_base;    // base address of x[] in mem
11       int y_base;    // base address of y[] in mem
12       int i;
13       int rval;
14
15       // create memory map for x[] and y[] and initialize mem
16       x_base = 0;
17       y_base = x_base + n;
18       for (i = 0; i < n; i++) {
19          mem.write(x_base + i, x[i]);
20          mem.write(y_base + i, y[i]);
21       }
22
23       // configure agp and macp
24       agp.load_program("dot_product.pgm");
25       agp.config(x_base, 1, 0, y_base, 1, 0, n, 0, 0);
26       macp.config(MAC, 1, 1);
27
28       // create connections between satellites
29       agp.connect(A);
30       mem.connect(A, unused, X);     // data input is unused
31       macp.connect(X, unused, O);    // Y input is unused
32
33       // run
34       for (i = 0; i < n; i++) {
35          agp.exec(); mem.exec(); macp.exec();
36          agp.exec(); mem.exec(); macp.exec();
37       }
38       // macp has written its results to O
39       rval = O.read().data();
40       agp.exec();    // last execution cycle of agp
41       return rval;
42    }
```

Figure 5.22: Intermediate Form Representation of Vector Dot Product

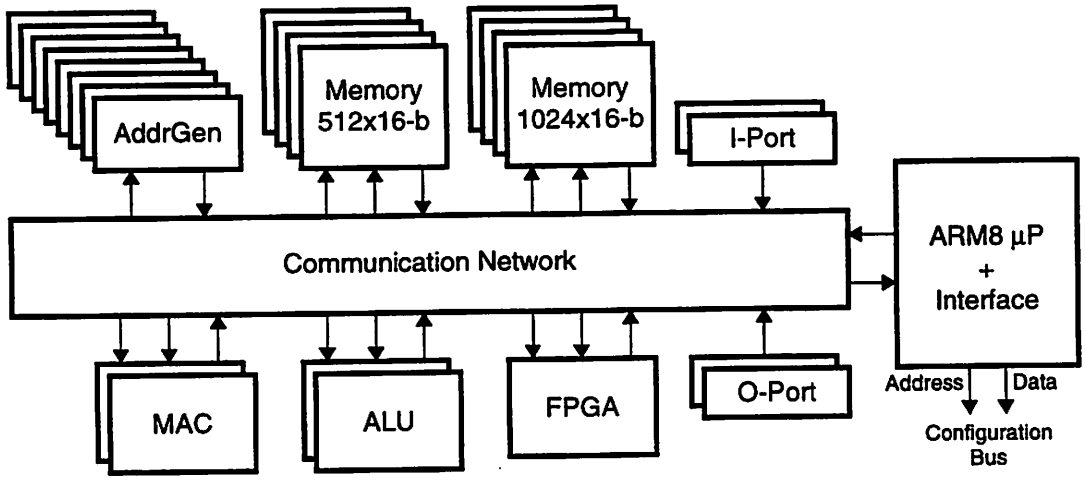


Figure 5.23: Block Diagram of the Maia Processor

models that are captured into the C++ objects representing the satellite processors and the communication network.

Further details of the Pleiades design methodology can be found in [123] and Marlene Wan's Ph.D. dissertation [124].

5.10 The Maia Processor

In this section, architectural design of Maia [125, 126], a Pleiades processor for CELP-based speech coding applications, will be presented. The Maia architecture was defined using the methodology outlined in Section 5.9. Figure 5.23 shows the block diagram of the Maia processor. The computational core of Maia consists of the following ensemble of satellite processors: 8 address generators, 4 512-word 16-bit SRAMs, 4 1024-word 16-bit SRAMs, 2 Multiply-Accumulate Units, 2 Arithmetic/Logic Units, a low-energy embedded FPGA unit, 2 input ports, and 2 output ports. To support CELP-based speech coding efficiently, 16-bit datapaths were used in the satellite processors and the communication network. The communication network uses a 2-level hierarchical mesh

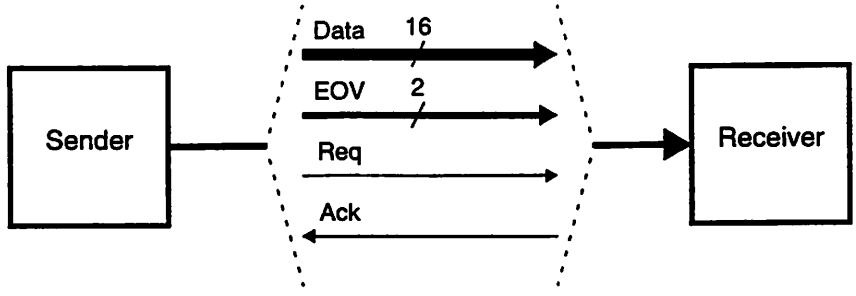


Figure 5.24: Bundled Signals of a Communication Network Channel

structure, as described in Section 5.5. To reduce communication energy, low-swing driver and receiver circuits are used in the communication network. Satellite processors communicate through the communication network using the 2-phase asynchronous handshaking protocol shown in Figure 5.17. Each link through the communication network consists of a 16-bit data field, a 2-bit EOV field, and a request/acknowledge pair of signals for data-driven control and asynchronous handshaking (see Figure 5.24). The EOV field can have one of three values: 0, 1, 2. As a result, the control mechanism used in Maia can support scalar, vector, and matrix data types. The I-Port and O-Port satellites are used for off-chip data I/O functions.

5.10.1 Control Processor Interface

The control processor in Maia is a custom implementation of the ARM8 microprocessor, a 32-bit RISC processor core [127]. The control processor was optimized for low-power operation and was designed to support dynamic scaling of the supply voltage.

The control processor communicates with the satellite processors through an interface module. The interface module performs the following functions:

- It allows the control processor to send and receive data tokens through the communication network, and as a result, the control processor can communicate with the

satellite processors through the communication network as just another satellite processor.

- It allows the control processor to configure the satellite processors and the communication network. The configuration bus is derived from the address/data/control bus of the control processor. It consists of a 16-bit address bus and a 16-bit data bus. The configuration bus can also be used by the control processor to write to and read from the SRAM satellites and the instruction memory of the address generators. The configuration state of all satellite processors, the contents of the SRAMs, and the contents of the instruction memories of the address generators are all part of the memory map of the control processor.
- It provides the control processor with the ability to reset the satellite processors and their handshake circuits by writing to the appropriate registers.
- It provides the control processor with the ability to initiate the execution of kernels and detect their completion. Kernels are initiated by sending request signals to the address generators. The control processor performs this function by writing to the appropriate registers in the interface module. Completion of kernels are signaled by the address generators through acknowledge signals, which are used to set special flag bits in the interface module that can either be polled by the control processor or can be used to interrupt the control processor.

Further details of the design of the interface module and its different operation modes can be found in Vandana Prabhu's Masters thesis [128].

5.10.2 Address Generator Processor

The address generator processor (AGP) is responsible for generating the address sequences that are needed to access data structures from the memory units while executing a kernel. The architecture of the AGP is based on a programmable datapath with a small

instruction memory. The AGP has a simple but flexible instruction set that allows the programmer to scan the elements of a vector or a matrix in complex but structured patterns. The instruction set of the AGP allows up to two levels of nesting in the address generation loop. The instruction set also supports multiplexing of two address streams onto the same communication channel. This allows the programmer to access simultaneously two different data structures that are stored in the same memory unit.

Execution of a kernel is initiated by the control processor by sending a request signal to the relevant AGP. The request signal triggers the execution of a pre-loaded program in the AGP. AGP programs are typically very short (just a few instructions at the most). Multiple programs can be stored in the instruction memory, which can store up to 16 instructions in the Maia implementation. The request signal that initiates the execution of an AGP program is accompanied by a data token that specifies which one of the pre-loaded AGP programs is to be executed. When the AGP executes a `halt` micro-instruction, and the last address token has been generated and sent, the AGP returns an acknowledge signal that can be used to interrupt the control processor.

The datapath of the AGP is shown in Figure 5.25. The Q register is the output of the AGP. An address token generated by the AGP contains a memory address and a control flag specifying the type of memory access, i.e., read or write. There are two address pointers: I0 and I1. If two multiplexed address streams are to be generated, then both pointers are used; otherwise, only one of them is used. The address pointers are loaded with initial values during configuration. Each address pointer has a *step* register and a *stride* register that contain signed values and are initialized during configuration. S0 is the step register for I0, and S1 is the stride register for I0. S2 is the step register for I1, and S3 is the stride register for I1. An address sequence is generated by repetitively adding the value of either the step register or the stride register to the address pointer, under program

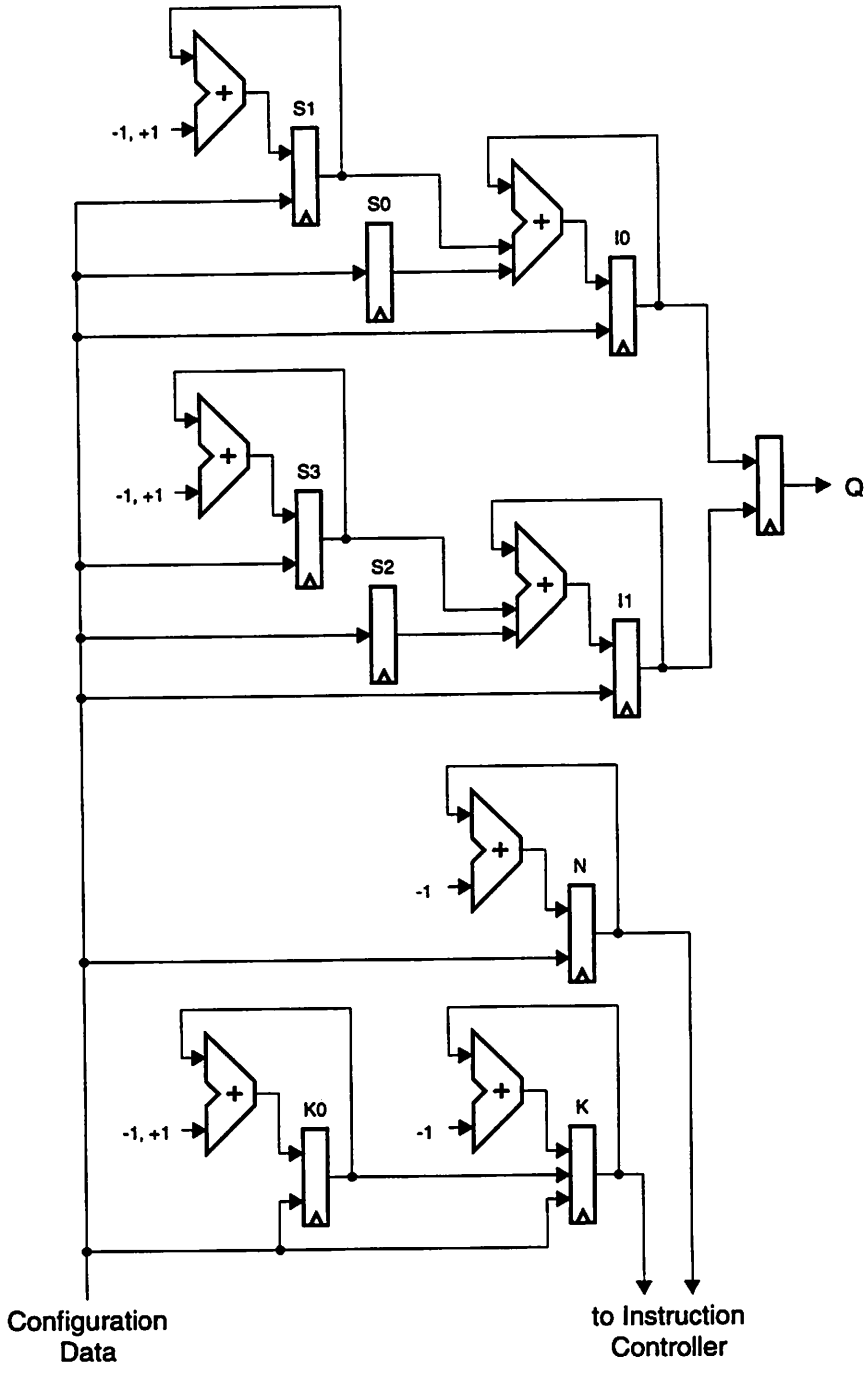


Figure 5.25: AGP Datapath

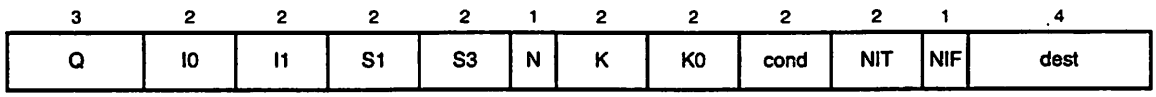


Figure 5.26: AGP Instruction Format

control. The value of a stride register can be incremented or decremented by one, under program control. The AGP datapath includes two loop index counters, N and K. If a two-level nested loop is needed, then K serves as the inner loop index counter, and N serves as the outer loop index counter; otherwise, N is used as the loop index counter. The loop index counters are initialized during configuration. They are always decremented by one, under program control, until they reach 0, which indicates the last iteration of a loop. K can also be loaded with K0. This option can be used to change the number of iterations of the inner loop at run-time, as K0 can be incremented or decremented, under program control. The EOV flags are inserted into the address tokens based on the values of the loop index counters. For a nested loop, during the last iteration of the outer loop, when N is 0, the last address token generated by the inner loop is marked with an EOV value of 2. For other iterations of the outer loop, the last address token generated by the inner loop is marked with an EOV value of 1. The EOV value attached to all other address tokens is 0.

Figure 5.26 shows the format of an AGP instruction (the width of each field is indicated above the field). The allowed operations of each field of an AGP instruction are listed in Table 5.1. The instruction to be executed next can be subject to an optional condition. If the instruction to be executed next is not subject to a condition, then the NIT (Next Instruction True) field is used to determine the next instruction. If the instruction to be executed next is subject to a condition, and the condition turns out to be true, then the NIT field is used to determine the next instruction; otherwise, the NIF (Next Instruction False) field is used. If the specified condition turns out to be true, then the Q, I0, I1, S1, S3, N, K, and K0 operations of the instruction are nullified, i.e., they produce no side effect.

Field	Explanation	Operations
Q	operation for Q register	I0 read, I0 write, I1 read, I1 write, no op
I0	operation for I0 register	add S0, add S1, no op
I1	operation for I1 register	add S2, add S3, no op
S1	operation for S1 register	increment, decrement, no op
S3	operation for S3 register	increment, decrement, no op
N	operation for N register	decrement, no op
K	operation for K register	decrement, load K0, no op
K0	operation for K0 Register	increment decrement, no op
cond	condition for next instruction	none, N == 0, K== 0
NIT	next instruction (unconditional or true conditional)	next, goto, halt
NIF	next instruction (false conditional)	next, here
dest	instruction address for goto	*<address>*

Table 5.1: Operations Executed by an AGP Instruction

Figure 5.27 illustrates an example of how an AGP can be programmed to produce a desired address sequence. In this example, the AGP is programmed to produce a multiplexed address stream to read two vectors stored in the same memory unit to calculate the dot product of the two vectors. Observe that the AGP program is expressed in pseudo-code using C syntax. More examples of AGP programs will be presented at the end of this chapter, where examples of kernel mappings will be presented.

5.10.3 Memory Units

The functionality of the memory unit is quite simple. A memory unit has three inputs: address (A), data in (DI), and data out (DO). An input address token on A includes a memory address, a read/write flag, and an EOV flag. The address input is typically generated by an address generator. If the address token specifies a read operation, then the

```
S = 0;
for (i=0; i<N; i++) {
        S += X[i]*Y[i];
}
```

Desired Sequence:

EOV	0	0	0	0	0	0	...	1	1
Address	X	Y	X+1	Y+1	X+2	Y+2		X+N-1	Y+N-1

X and Y are the base addresses of X[n] and Y[n], respectively.

AGP Program for Desired Sequence:

label	*Q*	*I0*	*I1*	*S1*	*S3*	*N*	*K*	*K0*	*goto*
L1	I0 read	+S0							(N==0) ? halt : next
	I1 read		+S2			-1			L1

Configuration:

I0 is loaded with X (X is the base address of X[n]).
S0 is loaded with +1.
I1 is loaded with Y (Y is the base address of Y[n]).
S2 is loaded with +1.
N is loaded with N (the length of the X[n] and Y[n]).

Figure 5.27: Example AGP Program

memory location specified by the address is read and sent to the DO output. The EOV flag of the address token is copied onto the EOV flag of the output data token. Thus, a memory unit preserves the type of data structure specified on the input address stream. If an address token specifies a write operation, then the data token on the DI input is written to the memory location specified by the address token. The memory unit has an additional function that allows the programmer to load a block of addresses specified by an address stream with zeros, without the need for a corresponding data stream that is needed by

memory write operations. Two memory sizes were chosen for Maia: 512-word and 1024-word. Both sizes can be used for all vector and matrix calculations. The 1024-word memories were selected for storing and manipulating the codebook structures that are commonly used in the CELP-based speech coding algorithms. The smaller memories consume less power and are favoured for most kernels that do not need the larger memories.

5.10.4 Multiply-Accumulate Unit

The core of the MAC satellite processor consists of a multiplier, followed by an accumulator. The MAC unit has two inputs, A and B, and one output, Q. The MAC unit performs one of two basic tasks: multiply and multiply-accumulate. The MAC unit has two pipeline stages in the Maia implementation. The MAC unit can perform one of four possible functions on the A and B streams:

- Scalar multiplication:

$$Q = A \times B \tag{5.1}$$

$$Q[i] = A[i] \times B[i] \tag{5.2}$$

$$Q[i][j] = A[i][j] \times B[i][j] \tag{5.3}$$

- Scalar-by-Vector Multiplication:

$$Q[i] = A \times B[i] \tag{5.4}$$

$$Q[i][j] = A[i] \times B[i][j] \tag{5.5}$$

- Scalar-by-Matrix Multiplication:

$$Q[i][j] = A \times B[i][j] \tag{5.6}$$

- Vector Dot Multiplication

$$Q = \sum_{i=0}^{N-1} A[i] \times B[i] \tag{5.7}$$

$$Q[i] = \sum_{j=0}^{N-1} A[i][j] \times B[i][j] \tag{5.8}$$

The dimensionality of the output data stream is derived from the EOV flags of the input data streams. The MAC unit automatically delimits its output data stream with the proper EOV flags. The MAC unit also has the ability to shift, round, and saturate the output result, as specified by the configuration state of the MAC unit. Instructions of the MAC processor can operate in a mode in which both input data streams arrive on the A input in a time-multiplexed fashion. This mode can be specified by the configuration state of the MAC unit. The B input is unused in this mode of operation.

5.10.5 Arithmetic/Logic Unit

The ALU processor performs a variety of arithmetic, logic, and shift operations. It has two inputs, A and B, an one output Q. It has three basic types of instructions:

- single-input scalar operations: absolute value and logical not. If the dimensionality of the input data streams is increased, then the executed functions will automatically become vector or matrix operations.
- two-input scalar operations: add, subtract, shift, min, max, compare, logical and, logical or, logical xor. Once again, these operations will automatically become vector or matrix operations, if the dimensionality of the input data streams is increased.
- two-input vector-to-scalar operations: accumulate, vector max, and vector min. Once again, these operations will automatically become two-dimensional vector operations, if the dimensionality of the input data streams is increased.

5.10.6 Embedded FPGA

The FPGA unit consists of a 4-by-9 array of 5-input, 3-output logic blocks. The design of the FPGA unit has been highly optimized for energy-efficient operation. The FPGA can have up to two input ports and an output port. The port behavior of the FPGA units is completely programmable and can be set by four of the 36 logic blocks. The

FPGA unit has two important functions that give the Maia architecture a great deal of flexibility:

- In addition to being able to implement the functions performed by the ALU processor (albeit at a higher cost), the FPGA can implement irregular bit-manipulation and arithmetic operations that cannot be supported by the MAC and ALU processors. The FPGA can also implement finite-state machines.
- The FPGA can be used to implement irregular address generation patterns that are not supported by the AGP instruction set. This can be done either in stand-alone fashion, or in conjunction with an AGP, in which case the FPGA performs a transformation function on the stream produced by the AGP. A good example of the latter is the bit-reversed addressing mode needed for performing FFT functions.

The details of the FPGA design is beyond the scope of this dissertation and can be found in [129] and in Varghese George's Ph.D. dissertation [130].

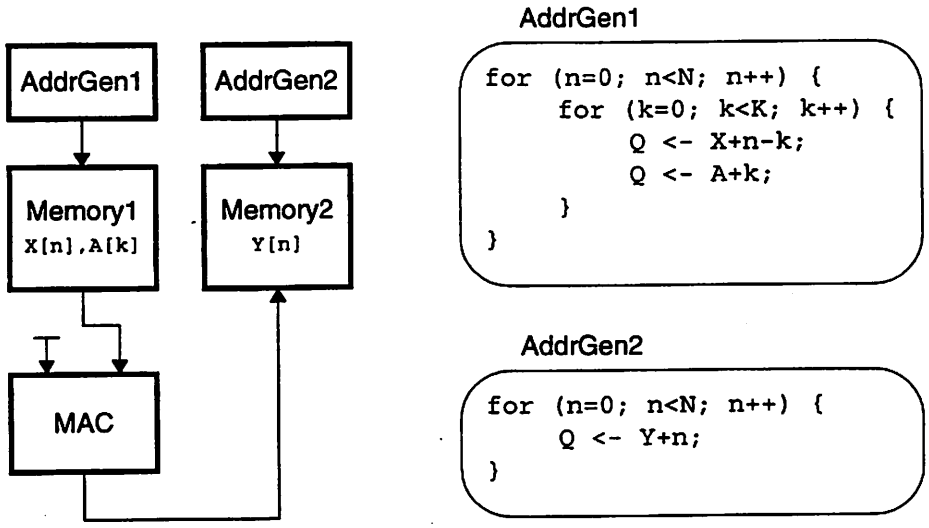
5.11 Algorithm Mapping Examples

In this section, we will present examples of how a kernel can be mapped onto satellite processors. Two examples will be considered. The first one is the ubiquitous Finite Impulse Response filter. The second one is the synthesis filter used in the VSELP speech coding algorithm.

5.11.1 FIR Filter

In this example, the response of a K-th order FIR filter to an input signal $X[n]$ $(0 \le n < N)$ is computed. The calculation performed by the kernel is

$$Y[n] = \sum_{k=0}^{K-1} A_k \cdot X[n-k] = A_0 \cdot X[n] + A_1 \cdot X[n-1] + \ldots + A_{K-1} \cdot X[n-K+1] \quad (5.9)$$



Note: Following C syntax, X, Y, and A refer to the base address (address of element 0) of X[n], Y[n], and A[k], respectively. Q is the output port of an address generator.

Figure 5.28: A Mapping for the FIR Kernel

where A_k $(0 \le k < K)$ are the coefficients of the FIR filter, and $Y[n]$ $(0 \le n < N)$ is the output of the filter. The FIR calculation can be specified in C-style pseudo-code as:

```
for (n=0; n<N; n++) {
     s = 0;
     for (k=0; k<K; k++) {
          s += X[n-k]*A[k];
     }
     Y[n] = s;
}
```

Figure 5.28 shows a mapping of the FIR kernel in which the X[n] and A[k] vectors reside in the same memory unit, and the Y[n] vector is required to be stored in another memory unit. During the first iteration of the outer loop of AddrGen1, the address sequence produced by the inner loop will result in the following sequence from Memory1:

X[0], A[0], X[-1], A[1], ... , X[-K+1], A[K-1]

label	*Q*	*I0*	*I1*	*S1*	*S3*	*N*	*K*	*K0*	*goto*
L1							K0		(N==0) ? halt : next
L2	I0 read	+S0							(K==0) ? E2 : next
	I1 read	·	+S2				-1		L2
E2		+S1	+S3			-1			L1

Configuration:

I0 is loaded with X (X is the base address of X[n]).
S0 is loaded with -1.
S1 is loaded with K.
I1 is loaded with A (A is the base address of A[k]).
S2 is loaded with +1.
S3 is loaded with -K+1.
N is loaded with N.
K0 is loaded with K.

Figure 5.29: Program for AddrGen1 of Figure 5.28

where data tokens with EOV=1 are underlined (EOV=0 for data tokens that are not underlined). During the last iteration of the outer loop of AddrGen1, the address sequence produced by the inner loop will result in the following sequence from Memory1:

X[N-1], A[0], X[N-2], A[1], ... , X[N-K], A[K-1]

where data tokens with EOV=2 are double-underlined. The MAC processor executes the dot multiplication operation on the incoming multiplexed data stream and produces the following sequence:

Y[0], Y[1], ... , Y[N-1]

The program running on AddrGen1 is shown in Figure 5.29, and the program running on AddrGen2 is shown in Figure 5.30.

label	*Q*	*I0*	*I1*	*S1*	*S3*	*N*	*K*	*K0*	*goto*
	I0 read	+S0				-1			(N==0) ? halt : here

Configuration:
I0 is loaded with Y (Y is the base address of Y[n]).
S0 is loaded with +1.
N is loaded with N.

Figure 5.30: Program for AddrGen2 of Figure 5.28

5.11.2 VSELP Synthesis Filter

The synthesis filter of the VSELP algorithm performs the following computation:

$$Y[n] = X[n] + \sum_{k=0}^{K-1} A_k \cdot Y[n-k-1] \tag{5.10}$$

where $X[n]$ $(0 \le n < N)$ is the input of the filter, A_k $(0 \le k < K)$ are the coefficients of the filter, and $Y[n]$ $(0 \le n < N)$ is the output of the filter. Observe that the output of the filter is a function of the past output of the filter, as well as the input of the filter. The calculation performed by the synthesis filter can be specified in pseudo-code as follows:

```
for (n=0; n<N; n++) {
     s = 0;
     for (k=0; k<K; k++) {
          s += Y[n-k-1]*A[k];
     }
     Y[n] = s + X[n];
}
```

Figure 5.31 shows a mapping of the synthesis filter kernel in which the `Y[n]` and `A[k]` vectors reside in the same memory unit, and the `X[n]` vector is stored in another memory unit. During the first iteration of the outer loop of AddrGen1, the following sequence of address tokens will be produced:

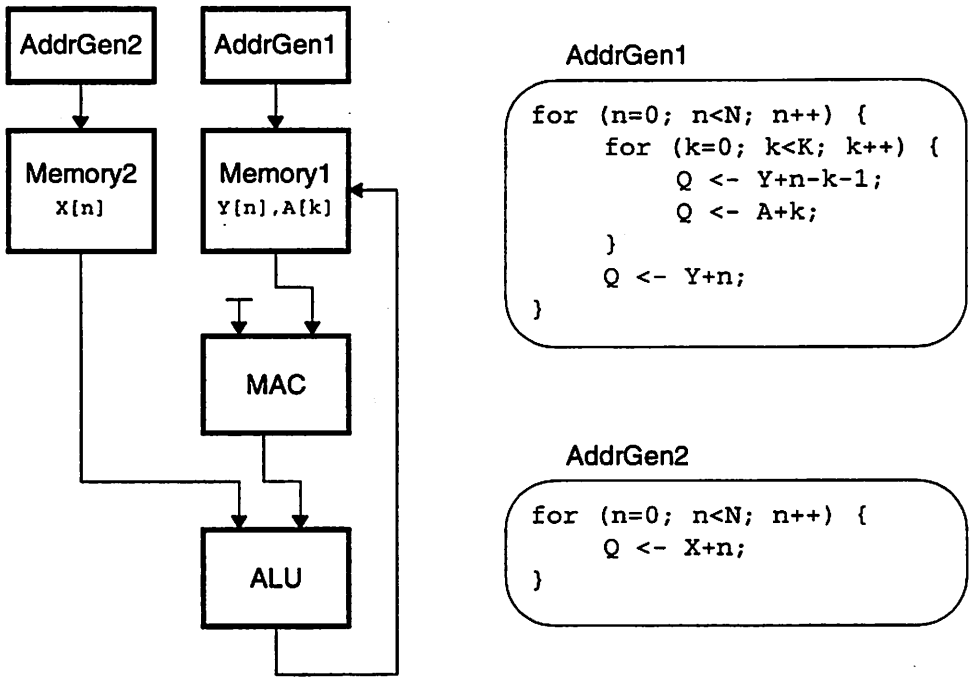


Figure 5.31: A Mapping for the VSELP Synthesis Filter

$$Y-1,\ A,\ Y-2,\ A+1,\ \ldots\ ,\ \underline{Y-K},\ \underline{A+K-1},\ \overline{Y}$$

where Y and A, following C syntax, are the base addresses of the Y[n] and A[k] vectors, respectively, and an overlined address token indicates that a memory write operation is to be performed with that address token. The last address token is used to write the output of the filter to Memory1. During the last iteration of the outer loop of AddrGen1, the following sequence of address tokens will be produced:

$$Y+N-2,\ A,\ Y+N-3,\ A+1,\ \ldots\ ,\ \underline{\underline{Y+N-K-1}},\ \underline{\underline{A+K-1}},\ \overline{\underline{Y+N-1}}$$

The MAC processor executes the dot multiplication operation on the incoming multiplexed data stream. The ALU executes the addition operation on its input data streams and produces the following sequence:

label	*Q*	*I0*	*I1*	*S1*	*S3*	*N*	*K*	*K0*	*goto*
L1							K0		(N==0) ? halt : next
L2	I0 read	+S0							(K==0) ? E2 : next
	I1 read	-	+S2				-1		L2
E2		+S1	+S3						next
	I0 write					-1			L1

Configuration:

I0 is loaded with Y-1 (Y is the base address of Y[n]).
S0 is loaded with -1.
S1 is loaded with K.
I1 is loaded with A (A is the base address of A[k]).
S2 is loaded with +1.
S3 is loaded with -K+1.
N is loaded with N.
K0 is loaded with K.

Figure 5.32: Program for AddrGen1 of Figure 5.31

label	*Q*	*I0*	*I1*	*S1*	*S3*	*N*	*K*	*K0*	*goto*
	I0 read	+S0				-1			(N==0) ? halt : here

Configuration:

I0 is loaded with X (X is the base address of X[n]).
S0 is loaded with +1.
N is loaded with N.

Figure 5.33: Program for AddrGen2 of Figure 5.31

Y[0], Y[1], ... , Y[N-1]

The program running on AddrGen1 is shown in Figure 5.32, and the program running on AddrGen2 is shown in Figure 5.33.

5.12 Summary

The Pleiades architecture template was presented in this chapter. The architecture template has been designed for energy-efficient implementation of domain-specific programmable processors for signal processing applications. Architectural design of Maia, a domain-specific processor for CELP-based speech coding applications, was presented. The key features of the Pleiades architecture template are:

- A highly concurrent, scalable multiprocessor architecture with a heterogeneous array of optimized satellite processors that can execute the dominant kernels of a given domain of algorithms with a minimum of energy overhead. The architecture supports dynamic scaling of the supply voltage.
- Reconfiguration of hardware resources is used to achieve flexibility while minimizing the overhead of instructions.
- A reconfigurable communication network that can support the interconnection patterns needed to implement the dominant kernels of a given domain of algorithms efficiently. The communication network uses a hierarchical structure and low-swing circuits to minimize energy consumption.
- A data-driven distributed control mechanism that provides the architecture with the ability to exploit locality of reference to minimize energy consumption. The control mechanism provides special support to handle the data structures commonly used in signal processing algorithms efficiently. The control mechanism also provides a framework for minimizing switching activity.

CHAPTER 6

Hardware Design of P1

In this chapter, hardware design of P1, the first Pleiades prototype, will be presented. The P1 prototype was designed and built to evaluate and verify the validity of the architectural concepts used in the Pleiades architecture template. An important objective of the P1 prototype was to build all the key components of the Pleiades architecture template and to integrate them into a complete implementation that could be used to explore the effectiveness of the Pleiades approach. Lessons learned from the P1 design were used to refine the Pleiades architecture template. These lessons were incorporated into the design of the Maia processor, which was described in Chapter 5. The P1 design was also used as an initial driver for the Pleiades design methodology.

6.1 P1 Hardware Organization

The block diagram of P1 is shown in Figure 6.1. The satellite processors employed in P1 include a multiply-accumulate (MAC) unit, two memory units, two address generators, two input ports (IPort), and one output port (OPort). All data and address tokens are

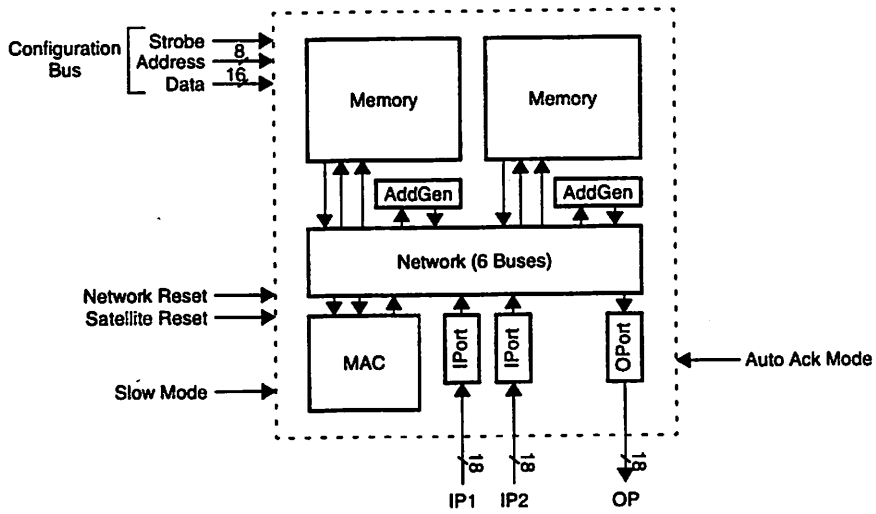


Figure 6.1: Block Diagram of P1

16-bit quantities and are handled by 16-bit datapaths in the satellite processors and 16-bit data buses in the communication network. P1 was not designed for a particular domain of algorithms, but its design was influenced by the properties of CELP-based speech coding algorithms. The chip can be used to implement the kernels shown in Figure 6.2.

P1 was fabricated in a 0.6-μm, 3.3-Volt CMOS technology through MOSIS [131]. The chip was designed to operate at a minimum cycle time of 50 ns with a 1.5-Volt supply voltage. The choice of the supply voltage was motivated by the desire to minimize power dissipation while maintaining acceptable performance. The chosen supply voltage results in an energy-delay product that is near the minimum for the CMOS technology used for P1 (see Figure 6.3). A plot of the P1 die is shown in Figure 6.4.

In order to measure and profile the power dissipation of the hardware modules used in P1, independent power supply pins were provided for the following circuit modules: the MAC unit, one of the memory units, one of the address generators, the network bus drivers of one of the IPort units, and the configuration bus drivers.

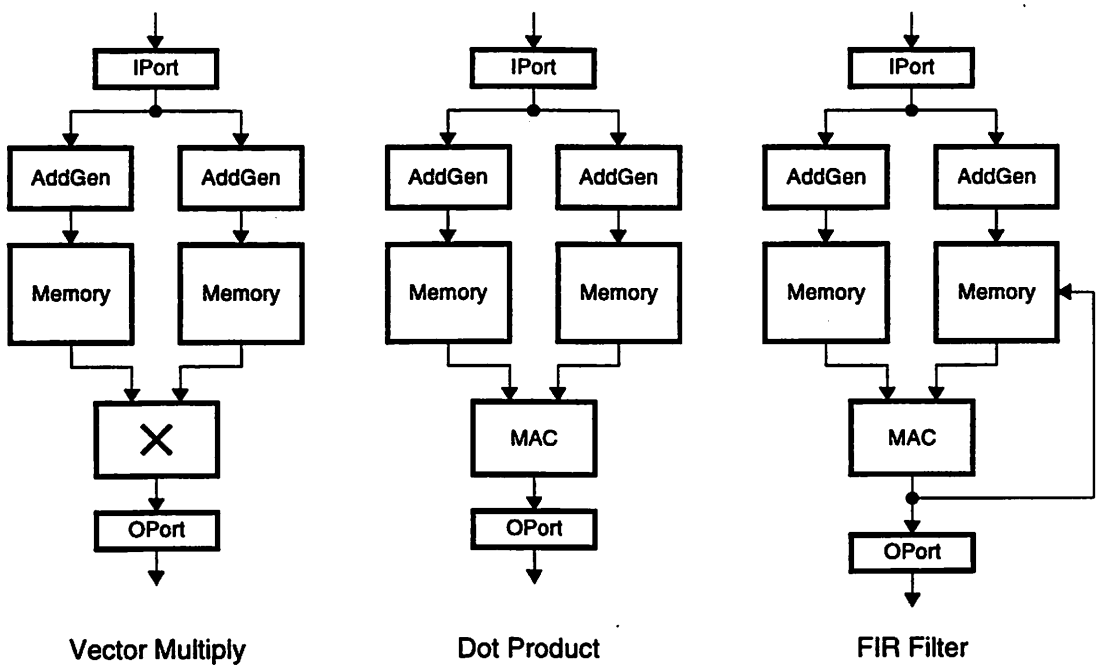


Figure 6.2: Kernels Supported by P1

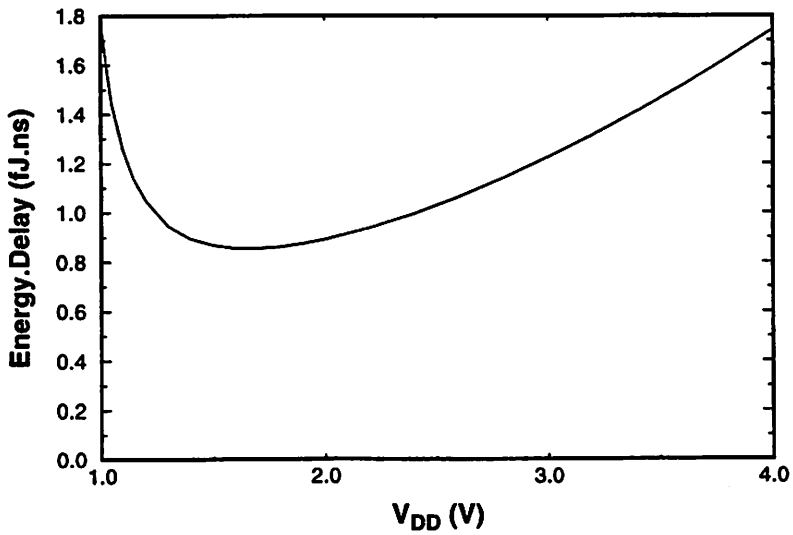


Figure 6.3: Energy-Delay Product vs. Supply Voltage

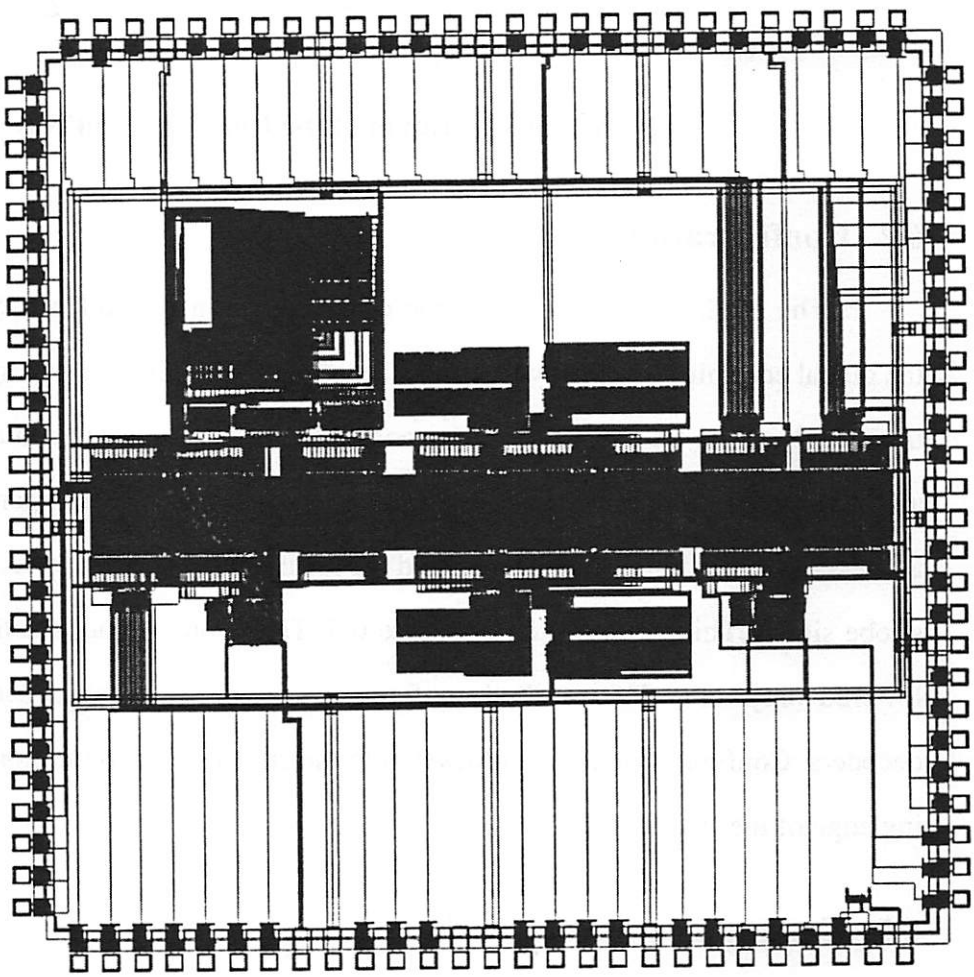


Figure 6.4: Die Plot of P1

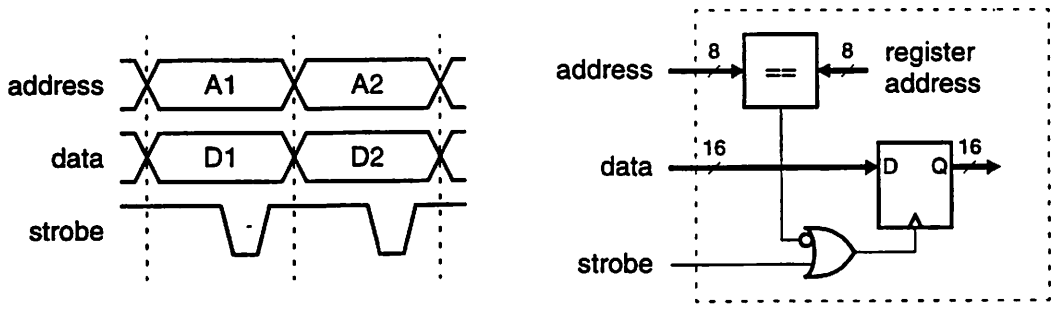


Figure 6.5: Operation of the Configuration Bus

6.2 Configuration Bus

The satellite processors and the communication network are configured through the global configuration bus, which consists of an 8-bit address bus to specify a configuration register, a 16-bit data bus to carry configuration information, and a strobe signal. For any given configuration register, the configuration address is compared to the assigned address of the configuration register, and the result of the comparison is used to qualify the strobe signal. This is illustrated in Figure 6.5. The strobe signal is active-low. It must be lowered only after the specified configuration address has been decoded by all local decoders. Configuration data is clocked into the specified configuration register at the rising edge of the strobe signal.

6.3 Communication Network

The communication network uses a full crossbar architecture with 6 19-bit buses, and as a result, any satellite input port can be connected to any of the satellite output ports by configuring the switches of the communication network. Satellite processors communicate across the network using the 2-phase asynchronous protocol (see Figure 5.17). Each bus of the communication network is 19 bits wide and contains the following signals: a 16-bit data bus, a request signal, and two acknowledge signals.

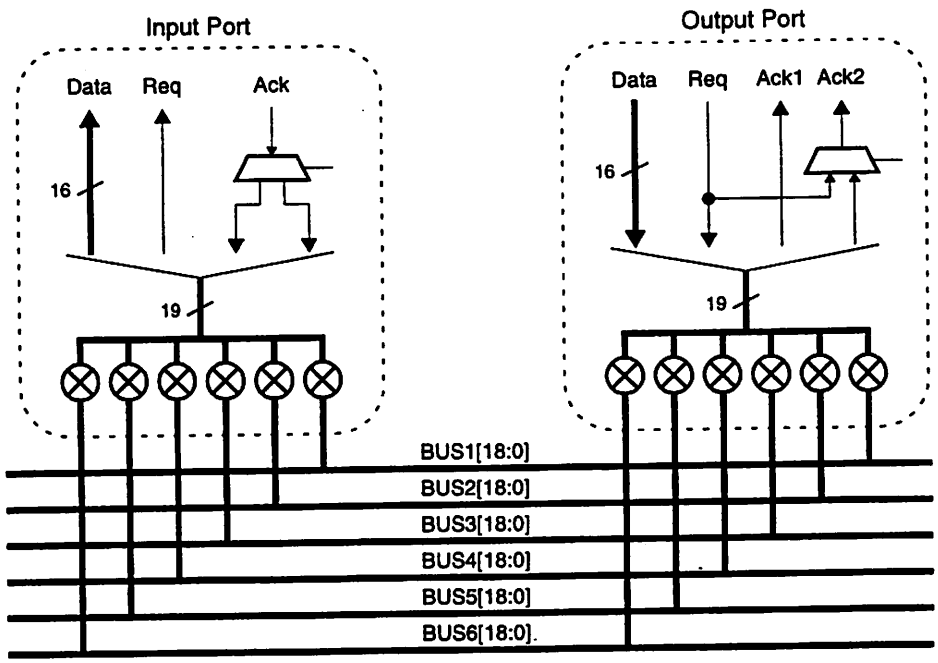
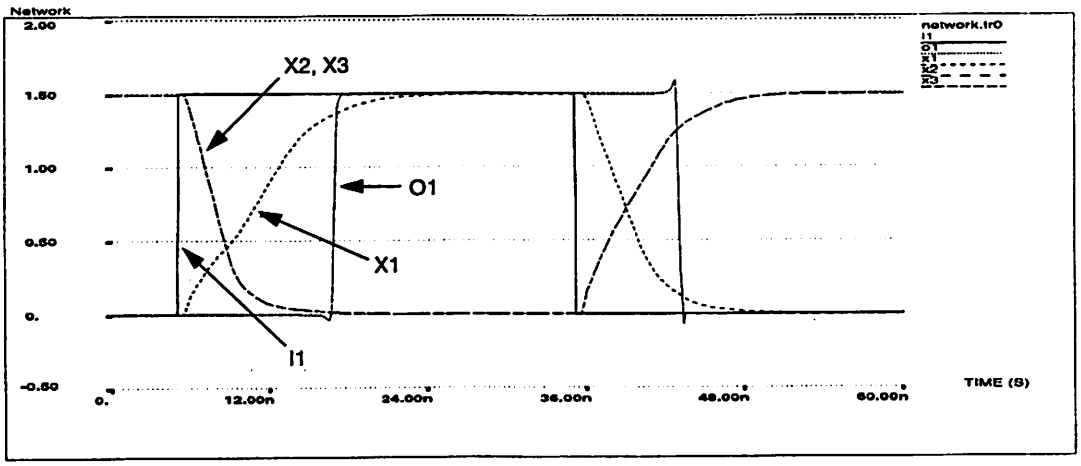


Figure 6.6: Port Structure of the Satellite Processors

The input and output ports of the satellite processors contain the communication network switches (see Figure 6.6). The switches are CMOS transmission gates and are controlled by the contents of the associated configuration registers. An output port of a satellite processor can have a maximum fanout of two, i.e., it can be connected to up to two different input ports. This is accomplished by using the second acknowledge signal provided in the buses of the communication network. The second acknowledge signal can be used to synchronize the output port with a second input port. If the fan-out of an output port is one, then the port is configured such that the second acknowledge signal is connected to the outgoing request signal, and every request automatically generates an acknowledge on the Ack2 signal (the internal handshake circuits of the satellite processors consider both acknowledge signals as active). The input ports also need to be configured to use one of the two acknowledge signals, as appropriate.



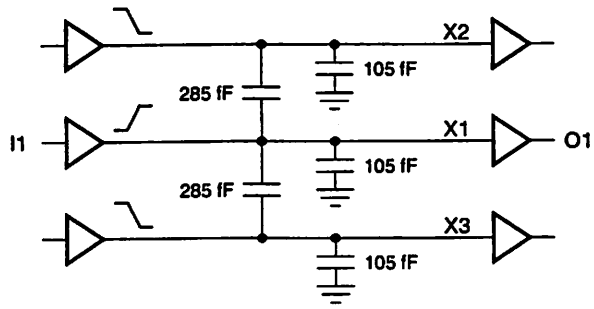


Figure 6.7: Waveforms for Communication Network (Worst-Case Coupling)

Worst-case delay across the communication network under worst-case coupling conditions is 11.2 ns. Figure 6.7 shows waveforms from a circuit simulation of the communication network.

6.4 I/O Ports

The IPort and OPort units (see Figure 6.1) are used to stream data tokens into and out of the chip. They communicate with the satellite processors through the communication network, and they behave as satellite processors. They communicate with off-chip circuits using the 2-phase asynchronous protocol with a 16-bit data bus, a request signal, and an acknowledge signal. The output port has an open-loop mode of operation, controlled by

the "Auto Ack Mode" pin (see Figure 6.1). When "Auto Ack Mode" is high, the output port does not wait for an acknowledge signal from off-chip circuits and immediately sends an incoming data token from the communication network to off-chip circuits. This mechanism was added for convenience during testing.

6.5 The MAC Unit

The MAC unit performs two basic functions: multiply and multiply-accumulate. The functionality of the MAC unit is chosen by its configuration state, which is stored in a single 16-bit register. The two input operands of the MAC unit are 16-bit signed integers. The multiply-accumulate function is used to compute the dot product of two input vectors. The length of the input vectors is specified by the configuration state of the MAC unit. The maximum vector size is 256. The MAC unit has a 40-bit accumulator, allowing it to accumulate at least 256 32-bit products without resulting in an overflow.

Figure 6.8 shows the block diagram of the synchronous functional core of the MAC unit. The clock signals of the MAC unit, CK1, CK2, and CK3, are generated by the asynchronous handshake controller of the MAC unit. The MAC unit has two pipeline stages. The multiplier design used in the MAC unit is based on the radix-4 modified Booth structure [132, 133] with a carry-save array to add the 8 partial products specified by the Booth encoder. The output of the carry-save array is in carry-save format and consists of a 32-bit sum vector and a 32-bit carry vector. The 40-bit output of the accumulator register, which is also in carry-save format, is added to the output of the carry-save array to produce the final result of the first pipeline stage. For a multiply operation, this result is loaded into the pipeline register clocked by CK3. For a multiply operation, the CK2 clock is inactive, and the 40-bit carry and sum vectors going into the carry-save adder block are forced to zero. For a multiply-accumulate operation, the result of the first pipeline stage is loaded only into the accumulator register (CK3 is inactive). However, for the last multi-

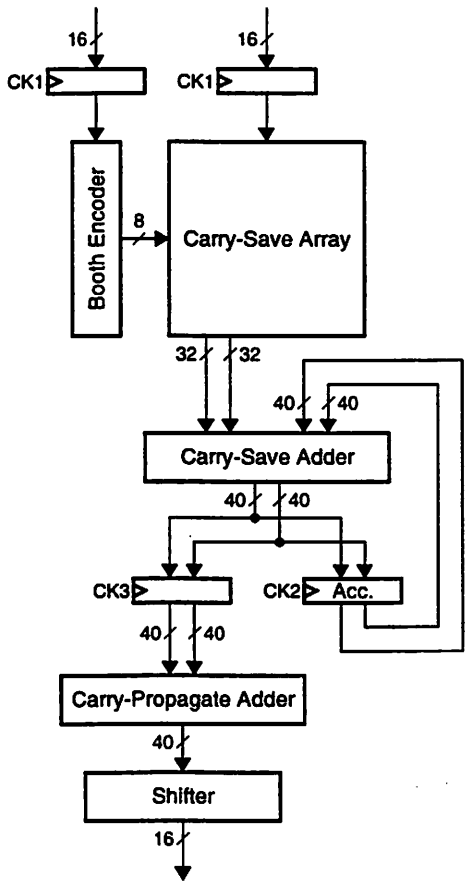


Figure 6.8: Block Diagram of the Functional Core of the MAC Unit

ply-accumulate operation of a vector dot product, the result of the first pipeline stage is also loaded into the pipeline register clocked by CK3. The carry-save result of the first pipeline stage is converted to 2's complement by the carry-propagate adder (CPA) block. The output of the CPA is shifted right by 0, 4, 8, 12, 16, 20, or 24 bit positions (specified by the configuration state of the MAC unit), and the least significant 16 bits of the shifted result form the output of the MAC unit.

Based on circuit simulation results, the cycle time of the functional core of the MAC unit is 39 ns and is determined by the first pipeline stage, i.e., the CSA stage. The CPA stage was not timing-critical and was built using a compact block carry-lookahead

structure with 8-bit carry-ripple blocks. The maximum delay through the CPA stage is 27 ns.

6.6 The Memory Units

The memory unit has three ports: an address input, a data input, and a data output. The core of the memory unit is a 256-word, 16-bit SRAM block. The SRAM is internally divided into two 128-word sub-blocks. To reduce access energy, only one of the sub-blocks is activated during an access, as specified by the most significant bit of the input memory address. The design of the SRAM provides a mode of operation that can be used to save energy during vector read operations, during which consecutive read operations access adjacent memory locations. Each row of memory cells stores two words. The addresses of the two words differ only in the least significant bit. The bit-slices of the two words are interleaved in the memory array such that two adjacent columns store the-same bit position of the two words (see Figure 6.9). Two such columns share a sense amplifier. For a read access, the bit lines are precharged first, then the cells of the selected row discharge the bit lines and the column specified by the least significant bit of the input address is selected. If the next read operation accesses the other column, then there is no need to precharge and discharge the bit lines again because those events have already occurred. All that is needed is to sense the previously discharged bit lines of the adjacent column. This type of access is called Precharge-Hold Access (PHA) and is controlled by a PHA signal that is part of the input address token provided by an address generator. The first cycle of a PHA access is exactly like a non-PHA, random-access cycle, but the second cycle of a PHA access takes less time and energy because there is no need to precharge and evaluate the bit lines.

An input address token includes an 8-bit address, a signal indicating the type of access (read or write), and a signal requesting the PHA mode. If the address token indi-

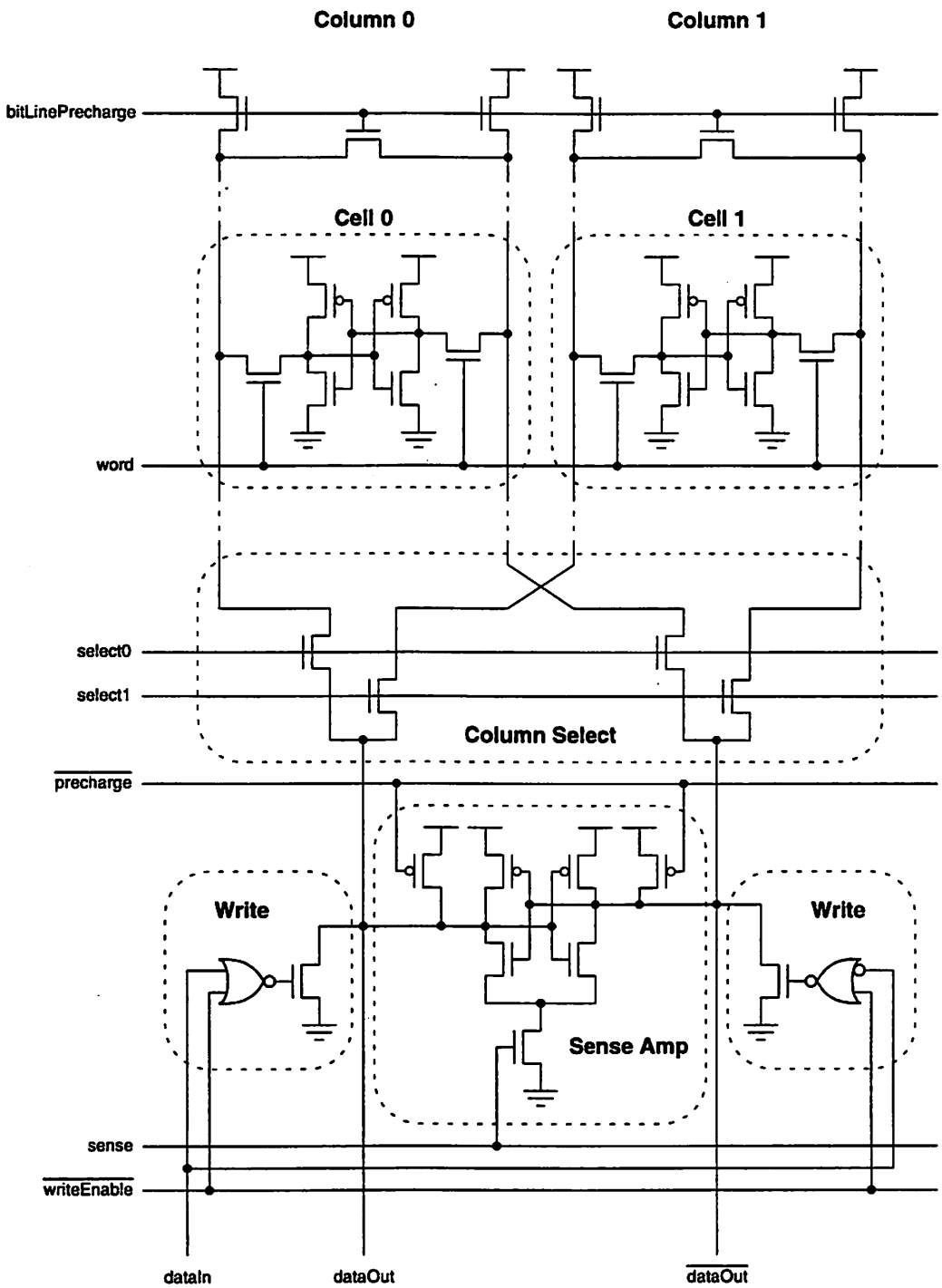


Figure 6.9: Bit-Line Structure of the P1 SRAMs

cates a read operation, then the specified memory location is read and sent to the data-out port of the SRAM. If a write operation is specified, then the incoming data token on the data-in port is written to the specified address. The clock signals of the synchronous SRAM core are provided by an asynchronous handshake controller that takes into account the operation mode of the SRAM to provide clock signals with appropriate timing.

6.7 The Address Generators

The main functionality of the address generator design used in P1 is to generate an address sequence to access a vector $X[n]$ $(0 \leq n < N)$ stored in a memory unit. The main objective of the design was to support the kernels shown in Figure 6.2. This is accomplished by counting up (or down) from an initial address `A1` to an end address `A2`. `A1` and `A2` are 8-bit values and are specified by the configuration state of the address generator, which also specifies the count step (+1 or −1). For testing and measurement purposes, the address generator can also be configured to generate a pseudo-random sequence counting from `A1` to `A2`.

Since the address tokens generated by the address generator are to be used by a memory unit, they must also provide a signal specifying the type of memory access (read or write) and a signal requesting a PHA read access. These two signals are controlled by the configuration state of the address generator and are bundled with an 8-bit address into a 16-bit token and transmitted through the communication network. The type of memory access specified by the configuration state can be *read*, *write*, or *read with last-write*. For the read mode, all outgoing tokens of an address sequence specify a read operation. For the write mode, all outgoing address tokens specify a write operation. For the read with last-write mode, all outgoing addresses, except the last one, i.e., `A2`, specify a read operation. The last address specifies a write. This mode can be used to implement the FIR filter kernel shown in Figure 6.2. The dot product of input vector `X[A1,...,A2-1]` and the

coefficient vector C[0,...,N-1] is stored back in X[A2], which is no longer needed to calculate the response of the filter, as X[A1] is the most recent input sample of the FIR filter, and X[A2-1] is the oldest required input sample, as determined by N, the length of the filter (N = A2 - A1). By repeating the kernel for X[A1-1,...,A2-2], the response of the FIR filter for the next input sample, i.e., X[A1-1], will be computed and stored in X[A2-1].

The address generator has an input port that can be used to request the generation of an address sequence. This is accomplished by sending an empty data token to the input port of the address generator. When the address generator has finished generating the specified address sequence, it returns an acknowledge signal on the input port and stops. To generate another address sequence, another request signal is needed. The address generator can also be triggered into generating a specified address sequence by writing to a special bit of the configuration state. The address generator has an additional operation mode, specified by the configuration state, in which it can repeat an address sequence in an infinite-loop mode after it has been triggered once. In the infinite-loop mode, the address generator will be stopped only by the global satellite reset signal.

The configuration state of the address generator is specified by two 16-bit registers. The first register contains A1 and A2, and the second register contains the bits specifying the operation mode of the address generator.

6.8 Chip Design Methodology

With a few exceptions noted below, all circuit blocks, including the top-level design of the chip, were implemented using a full-custom design methodology in the Cadence design environment [135]. The synchronous core of the address generator and all satellite controllers were specified in VHDL and synthesized using the Synopsys logic synthesis tool [134]. They were placed and routed using a standard-cell layout methodol-

ogy with the Cadence Cell Ensemble place-and-route tool. The logic design of the handshake circuits was done using a full-custom approach with special standard cells designed specifically for the handshake circuits. The layouts of the handshake circuits were rendered using a standard-cell methodology. The functionality of the chip was verified by logic and circuit simulations. Detailed critical path simulations were performed with the HSpice circuit simulator [136]. Block- and chip-level simulations were performed with PowerMill [134], which was also used to determine the power dissipation of the chip and its various sub-blocks.

6.9 Measurement Results

A custom circuit board was built to test and characterize the P1 chips, which were packaged in a 120-pin ceramic PGA package. Input vectors were provided using a logic analyzer. The exact same vectors were used with PowerMill simulations to make a direct comparison of measured and simulated energy and delay parameters. All measurements were done at room temperature, using a 1.50-Volt supply voltage. The results of the measurements and the simulations are listed and compared in Tables 6.1 and 6.2. For these measurements, the input ports of the satellite processor in question were driven by the IPort units, and the output of the satellite processor was sent to the OPort unit, which was operated in the open-loop mode, as described in Section 6.4. The IPort units provided data tokens to the satellite processor under test as soon as the satellite processor had acknowledged the receipt of a new input token. Thus, the cycle-time of the test setup was limited by the cycle-time of the satellite processor under test. Observe that this cycle time includes the round-trip delay of the communication network plus the input handshake delay of the OPort unit. This extra delay could not be directly measured but was estimated to be about 24 ns. Cycle-time measurements were performed by measuring the period of the input acknowledge signal of the satellite processor in question. This was possible because the

Circuit Module		Energy (pJ/cycle)		$\frac{E_{measured}}{E_{simulated}}$
		Simulated	Measured	
Address Generator Satellite (random mode)		8.1	7.3	0.90
MAC Satellite (multiply)	zero input	11.9	10.5	0.88
	random input	92.2	72.4	0.79
MAC Satellite (multiply-accumulate)	zero input	14.1	11.6	0.82
	random input	116.5	95.1	0.82
SRAM Satellite (read)	random data	33.7	32.4	0.96
SRAM Satellite (PHA read)	random data	27.9	25.7	0.92
SRAM Satellite (write)	random data	25.8	23.5	0.91
Network Channel	random data	8.3	6.8	0.82

Table 6.1: Energy Measurement and Simulation Results

Circuit Module		Cycle Time (ns)		$\frac{T_{measured}}{T_{simulated}}$
		Simulated	Measured	
Ring Oscillator	inverter (51-stage)	35.2	40.0	1.14
	delay cell (15-stage)	39.1	49.8	1.27
Address Generator Satellite		40.0	47.3	1.18
SRAM Satellite		35.6	42.4	1.19
MAC Satellite		70.8	87.4	1.23

Table 6.2: Cycle-Time Measurement and Simulation Results

Circuit Module	Energy (pJ/cycle)		$\frac{E_{measured}}{E_{simulated}}$
	Simulated	Measured	
Address Generator Satellite	5.0	4.4	0.88
SRAM Satellite (PHA read)	27.8	25.4	0.91
MAC Satellite (multiply-accumulate)	107.1	90.5	0.85
Network Channel (A input of MAC)	9.1	7.5	0.82
Total (chip core)	207.1	179.1	0.86

Table 6.3: Dot Product Results

request signal, the acknowledge signal, and the least significant data bit of BUS0 were driven off-chip and could be monitored by an oscilloscope. Energy measurements were performed by measuring the current through the supply pin of the circuit module in question with a current meter.

Energy measurement and simulation results for the dot product kernel, as shown in Figure 6.2, are listed in Table 6.3. The simulated cycle time for the dot product kernel was 71.4 ns. The measured cycle time of the kernel, based on the period of the input acknowledge signal of the MAC satellite processor, was 88.3 ns, i.e., 1.24 times the simulated value.

6.10 Discussion

A number of important lessons were learned during the design and evaluation of the P1 prototype. These lessons were used to refine the Pleiades architecture template and resulted in a number of significant improvements that were utilized in the Maia processor:

- The functionality of the address generator architecture used in P1 was limited to simple sequential access of the elements of a vector. One limitation was that the

count step used by the address generator was either +1 or −1 . Another limitation was that the count step could not be changed at run-time. Yet another limitation was that only one level of nesting was allowed in the address generation loop. A further limitation was that only a single data structure could be accessed by an address generator, as two address streams could not be multiplexed onto the same address generator. As a result, a second required data structure had to be stored in a separate memory and accessed by a separate address generator. All of these shortcomings pointed towards a solution that would provide more flexibility with a programmable datapath under the control of a small, simple instruction set designed specifically for generating address sequences. This resulted in the address generator architecture that was developed for the Maia processor (see Section 5.10.2).

- The MAC unit of P1 had to be configured to know the length of its input vectors. This meant that the MAC unit had to contain a replica of the loop index counter of the address generator. Not only does this approach waste area and energy, it is not clear how it can be extended to handle data structures more complex than simple vectors. This deficiency led to the development of the data-structure control mechanism with EOV flags, as described in Section 5.7.1.
- The satellite processors and the communication network had to be configured first before they could be used for any purpose. This meant that the overhead of reconfiguration cycles reduced the performance of the design. This shortcoming led to the overlapped reconfiguration and execution techniques that were discussed in Section 5.6.
- The delay of the communication network increased the cycle time of the design and reduced throughput significantly. This was exacerbated by the asynchronous timing scheme which required a round trip of request/acknowledge signals across

the network. To reduce this overhead, transfer of data tokens through the communication network should be a pipeline stage. This approach was used in Maia.

- The energy consumption of the communication network was responsible for about 15% of the total energy of the dot product kernel. To reduce this overhead, low-swing driver and receiver circuits were used in the Maia communication network.
- It turned out that the delay lines that were used by the satellite handshake controllers accounted for as much as 8% of the energy consumption of the satellite processors. This led to the development of more efficient delay lines for the Maia processor, where the energy consumption of the delay lines was kept to below 1% of total satellite processor energy. A further limitation of the P1 design was that the delay of the delay lines was fixed. To tune the delay of the delay lines to the minimum required for proper operation and hence maximize performance, the delay lines of the Maia processor could be adjusted via the configuration state of the satellite processors.

CHAPTER 7

Evaluation of the Pleiades Approach

In this chapter, the Pleiades architecture will be evaluated. Benchmark results comparing the Pleiades architecture to other programmable architectures will be presented and discussed. Two sets of comparisons will be presented. The first set of comparisons will be based on results from the P1 prototype. The second set of comparisons will be for the Maia processor.

7.1 P1 Case Study

In this section, results from the P1 prototype will be used to compare the Pleiades architecture to a variety of programmable architectures that are commonly used to implement signal processing algorithms [137]. The kernels that were used as benchmarks represent three of the most commonly used DSP algorithms: the Finite Impulse Response filter (FIR), the Infinite Impulse Response filter (IIR), and the Fast Fourier Transform (FFT). We will first present and discuss the programmable architectures that were considered in this study, and we will explain how they were used to implement and evaluate the benchmark kernels. Next, we will discuss the methodology that was used to normalize energy

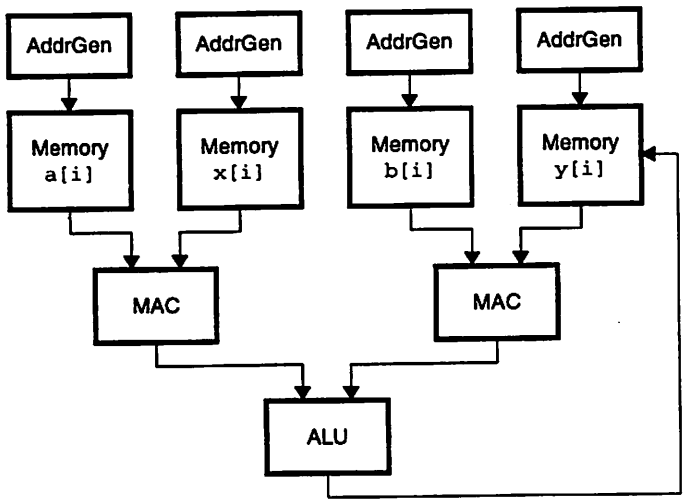


Figure 7.1: Pleiades Mapping for the IIR Kernel

and delay parameters of the studied architectures, so that comparisons could be done in a fair and uniform manner. Next we will present the results of the comparisons for each benchmark.

7.1.1 Pleiades

The Pleiades architecture was evaluated using the results of the P1 prototype. The FIR benchmark could be readily evaluated with the P1 design, as the FIR kernel was directly supported by P1. Since P1 does not have the hardware resources to implement the IIR and FFT benchmark kernels directly, these kernels were evaluated by extrapolating from P1 results.

The IIR benchmark was evaluated on Pleiades using the mapping shown in Figure 7.1. This mapping implements the IIR benchmark kernel used in this study:

$$y[n] = \sum_{i=0}^{4} a_i x[n-i] + \sum_{i=1}^{4} b_i y[n-i] \tag{7.1}$$

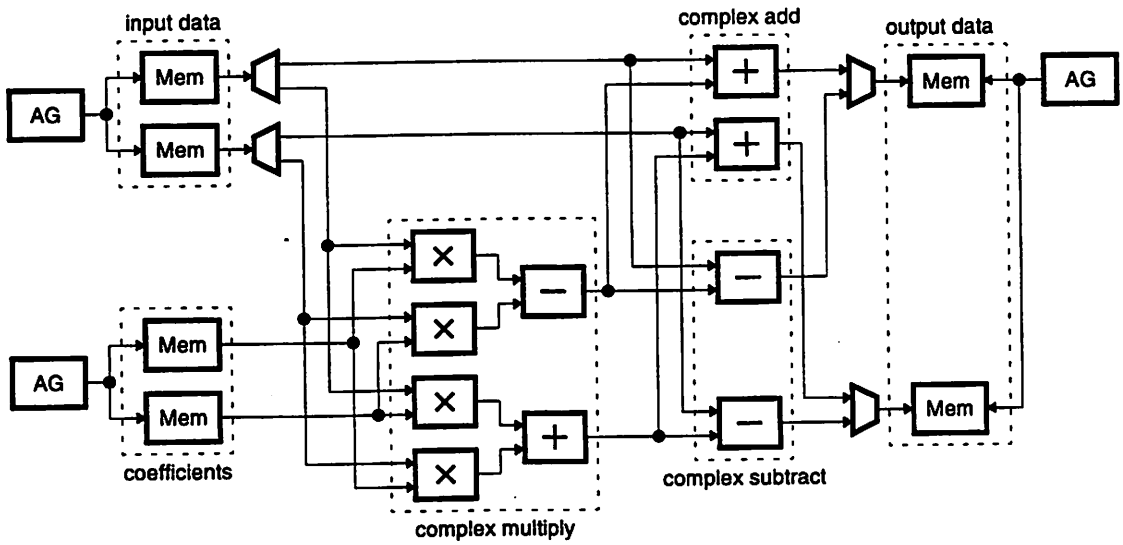


Figure 7.2: Pleiades Mapping for the FFT Kernel

Except for a slight modification to the address generators, the mapping of Figure 7.1 uses the same hardware resources that are available in P1. As a result, the IIR benchmark could be evaluated in a straightforward manner using the energy and delay models that were created for the hardware blocks used in P1.

The FFT benchmark used in this study was a 16-point, radix-2, decimation-in-time algorithm. To evaluate the FFT benchmark, a Pleiades-style processor was designed and simulated at the register-transfer level in VHDL. The performance and energy consumption of this processor were estimated using the energy and delay models from P1.

Figure 7.2 shows the hardware mapping for computing one stage of the FFT algorithm on the Pleiades processor. The design consists of 3 address generators, 6 memories, 4 multipliers, 6 ALUs, 2 splitter blocks, 2 merger blocks, and 23 buses. Note that a splitter block splits an incoming data stream into two output streams such that two consecutive input tokens are directed to different output streams. Similarly, a merger block merges two input data streams into a single output stream such that two consecutive output tokens are

taken from different input streams. An N-point FFT involves a total of $\log_2 N$ stages. Each stage involves $N/2$ butterfly calculations. Each butterfly calculation consists of a complex multiplication, a complex addition, and a complex subtraction. In the mapping shown in Figure 7.2, a single FFT butterfly is computed in each cycle. For successive stages of the FFT algorithm, the roles of the input and output data memories are exchanged, so that the output of the last stage becomes the input of the current stage.

The FFT processor uses the SRAM design used in P1. The multiplier design is similar to the MAC design in P1. As a result, the energy and delay models of the corresponding P1 blocks were used to evaluate the FFT benchmark. The delay and energy of the other blocks used in the FFT processor were estimated by synthesizing, placing, and routing their computational cores and simulating them with PowerMill using random data. The power of the communication network was estimated by extrapolating the communication network design used in P1. The estimates of network power are somewhat pessimistic since they assume random data. Actual signal data have temporal correlations that reduce switching activity. Furthermore, the network was assumed to be a full crossbar, but that was not necessary, and further savings in energy can be achieved by using the hierarchical mesh structure that was developed for the Maia processor. Table 7.1 shows the energy profile for a single FFT stage. The cycle time of the design is determined by the multiplier. Thus, ignoring configuration time, the time for one stage of a 16-point FFT is

$$T = 71.4 \text{ ns} \times \frac{N}{2} = 571 \text{ ns} \tag{7.2}$$

where $N = 16$. The power dissipation for the FFT algorithm is

$$P = \frac{13.3 \text{ nJ}}{571 \text{ ns}} = 23.3 \text{ mW} \tag{7.3}$$

Hardware Resource	Energy/Cycle (pJ)	Resource Count	Cycle Count	Energy/Stage (pJ)
AG (data)	13.1	2	16	419
AG (coefficients)	4.7	1	8	38
SRAM (read, data)	33.7	2	16	1078
SRAM (read, coefficients)	33.7	2	8	539
SRAM (write)	25.8	2	16	826
Multiplier	84.5	4	8	2704
ALU	12.1	6	8	581
Split	7.7	2	8	123
Merge	10.1	2	8	162
Network	26.8	9	16	3859
Network	26.8	14	8	3002
Total				13,331

Table 7.1: Energy Profile for 16-Point FFT Stage on Pleiades

7.1.2 The StrongARM Microprocessor

General-purpose microprocessors represent the ultimate in flexibility and are ubiquitously used to implement a wide variety of computational tasks. The StrongARM architecture was chosen as a reference because it represented the state-of-the-art in low-power, high-performance general-purpose microprocessor design. We evaluated the SA-110 microprocessor, a 32-bit, load/store RISC design with a Harvard architecture, a 16KB instruction cache, a 16KB write-back data cache, a write buffer, and a memory management unit on a single chip [138]. The SA-110 microprocessor has a multi-cycle multiply instruction.

The chip that we evaluated was implemented in a 0.35-μm CMOS technology and ran at 169 MHz with a 1.5-V supply voltage. To measure the energy consumption of a

benchmark kernel, we placed the code fragment for that kernel inside an infinite loop and measured the average current drawn by the microprocessor core while executing that loop. The StrongARM evaluation board that was used [139] had a voltage regulator that supplied power exclusively to the core of the SA-110 microprocessor (the I/O circuits of the chip were powered by a separate power source). By inserting a current meter in series with the output of this regulator we could measure the current drawn by the core while executing a given benchmark kernel. Both on-chip caches, the MMU, and the write-buffer were enabled and were included in the measurements. All kernels fit completely in the on-chip caches, so there was no off-chip memory traffic while executing the benchmark programs. The number of cycles spent executing a given kernel was obtained from the StrongARM emulator. All benchmark kernels were written in the C programming language and were compiled into assembly code using the ARM C compiler.

7.1.3 The Texas Instruments Programmable Signal Processors

A wide variety of DSP systems are designed with programmable digital signal processors. These processors are similar to general-purpose microprocessors but have extra instructions and addressing modes that improve their performance for DSP algorithms. An overview of programmable signal processor architectures was presented in Section 4.4. For the P1 case study, we chose two commonly used processors from Texas Instruments: the TMS320C2xx and the TMS320LC54x [140]. The TMS320LC54x is an advanced signal processor that was designed specifically for low-power operation.

The TMS320C2xx is a 16-bit, fixed-point processor that has on-chip instruction and data memories, a Harvard architecture, and a single accumulator. There is only one data bus in the TMS320C2xx design, but the instruction bus can be used to feed a second data stream into the arithmetic units. The chip we evaluated was fabricated in a 0.72-μm, 5.0-V CMOS technology and ran at 20 MHz with a 3.0-V supply voltage.

TMS320LC54x is a 16-bit fixed-point signal processor that has on-chip instruction and data memories, an enhanced Harvard architecture with three data buses, and two accumulators. In addition, it includes instructions that execute parallel operations. For example, the parallel-store-multiply instruction executes store and multiply in a single cycle (the TMS320C2xx lacks this capability). The chip that we analyzed was fabricated in a 0.6-μm, 3.3-V CMOS technology and ran at 40 MHz with a 3.0-V supply voltage.

Starting with assembly programs published by Texas Instruments (TI) in their application reports [141, 142, 143, 144], a set of benchmark programs were written in assembly language. The following programs were written: a 16-bit, 5-th order FIR filter; a 16-bit, 4-th order, direct-form IIR filter; and a 16-bit, 16-point complex FFT stage. All of these programs included initialization sections that were excluded for performance and power calculations. A 3.0-V supply voltage was assumed for these calculations.

Energy values were calculated by adding the contributions of all instructions in a kernel using instruction-level energy consumption data published by TI [145, 146]. It should be noted that this method produces somewhat optimistic results because it ignores inter-instruction effects that can slightly increase the energy consumption of an instruction [147]. The same method was used to calculate the number of cycles spent executing a kernel. An example of this process is shown in Figure 7.3.

7.1.4 The Xilinx XC4003A FPGA

Field-Programmable Gate Arrays have recently been used to implement a variety of high-throughput DSP applications that are beyond the reach of conventional signal processors. FPGAs are fully flexible and can be programmed to implement any algorithm, but they have a much finer grain of programmability than microprocessors and programmable signal processors, and as a result, they can incur large area and energy overheads. An

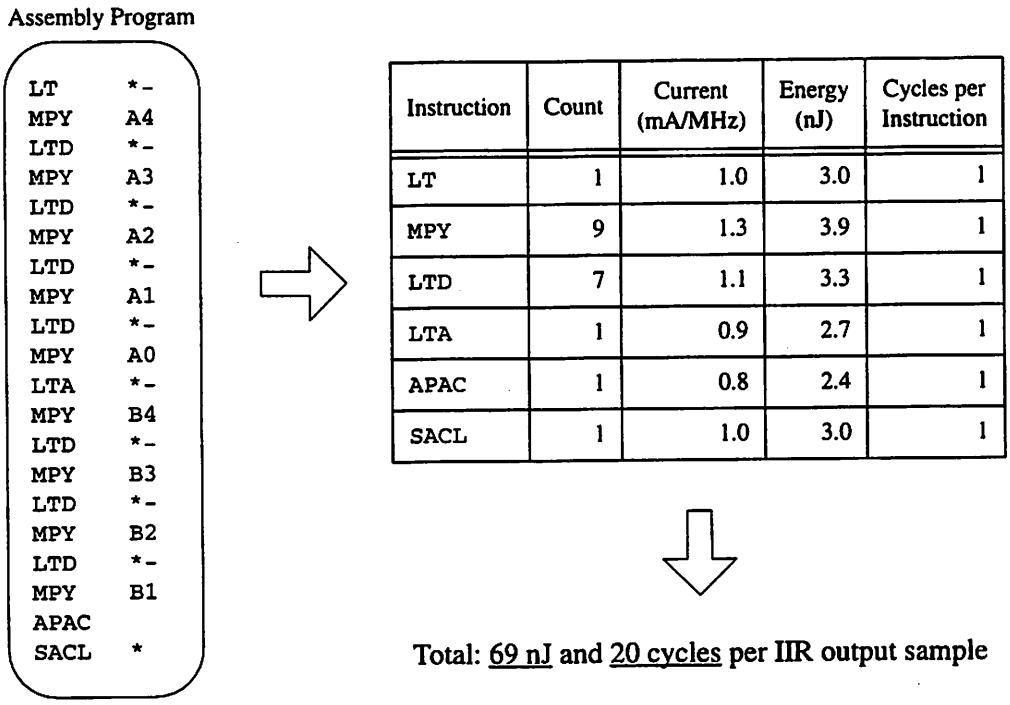


Figure 7.3: Instruction-Level Energy Calculation Example (IIR on TMS320C2xx)

overview of FPGA architectures was presented in Section 4.9. The FPGA device chosen for this study was the Xilinx XC4003A, a member of the widely-used XC4000 family of SRAM-based FPGAs from Xilinx [87]. The XC4003A has an equivalent logic capacity of 3000 gates. It contains 100 CLBs and 360 flip-flops. Each CLB consists of two 4-input LUTs and dedicated carry-logic that can be used to speed up arithmetic operations significantly.

The Xilinx evaluation board that was used for this study included a XC4003A chip. Since the XC4003A was too small for the larger benchmark kernels used in this study, smaller versions of those kernels were implemented and the obtained results were extrapolated. An 8-bit, 5-tap FIR filter with constant coefficients was implemented on the evaluation board and its energy consumption was measured directly. The measurements were then extrapolated to obtain energy values for a 16-bit filter. Since filter coefficients

were constant, add-and-shift multipliers were used in the design. This approach consumes much less energy than the general-purpose multipliers used in the other architectures, including P1. The FIR design is fully pipelined and produces an output sample every cycle.

For the IIR benchmark, an 8-bit IIR biquad section was mapped onto the XC4003A and its energy consumption was evaluated with an energy modeling tool for Xilinx FPGAs developed by Eric Kusse [90]. The input netlists for this analysis were created using the Hyper synthesis system [148].

7.1.5 Normalization of Results

The reference architectures that were considered in this study were implemented in different fabrication technologies, and they had different operating supply voltages. To make a meaningful comparison, the energy and delay metrics of these architectures had to be normalized to a common reference. We chose to normalize all figures of merit to the 0.6-μm, 3.3-V CMOS process that was used to implement P1. Recall that P1 was designed to operate with a 1.5-V supply voltage, and all energy and delay values are calculated for a 1.5-V supply voltage.

Switched capacitance is assumed to scale with gate capacitance and is normalized according to

$$C \propto \frac{A}{T_{ox}} \propto \frac{L^2}{T_{ox}} \tag{7.4}$$

where A is the gate area, L is the minimum channel length, and T_{ox} is the gate oxide thickness. T_{ox} was assumed to be proportional to the native supply voltage of a given process. Normalized energy is then computed using $E = CV_{DD}^2$ with $V_{DD} = 1.5$ V.

Processor	L_{min} (μm)	T_{ox} (nm)	V_{th} (V)	Native V_{DD} (V)	V_{DD} (V)	Normalization Coefficients	
						Capacitance	Delay
Pleiades	0.60	9	0.70	3.3	1.5	1.00	1.00
StrongARM	0.35	6	0.35	1.5	1.5	1.96	4.71
TMS320C2xx	0.72	14†	0.70†	5.0	3.0	1.08	1.37
TMS320LC54x	0.60	9†	0.70	3.3	3.0	1.00	1.97
XC4003A	0.60	14†	0.70†	5.0	5.0	1.56	2.67

Note: items marked with † are estimated values.

Table 7.2: Process Data and Normalization Coefficients

Delay is normalized according to

$$T = \frac{CV_{DD}}{I} \propto \frac{L^2 V_{DD}}{(V_{DD} - V_{th})^{1.3}} \tag{7.5}$$

where V_{DD} is the supply voltage, C is the load capacitance, I is the MOSFET saturation current, and V_{th} is the threshold voltage. Process parameters for all architectures are listed in Table 7.2.

For the StrongARM microprocessor, the low value of V_{th} results in large leakage currents. This leakage current is responsible for 20mW of maximum power dissipation when the processor is in the idle mode. This value was subtracted from the measured power dissipation values for capacitance calculations. Since this value is somewhat optimistic, it produces results that are favorable to the StrongARM processor.

As mentioned earlier, the XC4003A chip was not large enough to implement all benchmarks. 8-bit adders were implemented on the FPGA, and the associated capacitance values were multiplied by a factor of 2 to extrapolate the results for a 16-bit design. The

cycle time of the 8-bit design was multiplied by 2 to obtain results for a 16-bit design. Observe that these extrapolation factors are optimistic and produce results that are favorable to the XC4003A. The FPGA implementation of the FIR filter computes 5 taps concurrently, in a single cycle, so total energy was divided by 5 to compute the energy per tap value. For the IIR benchmark, the capacitance of an IIR biquad section, implemented on the XC4003A, was doubled to account for a 4-th order IIR filter.

7.1.6 Benchmark Results

Comparison results for the FIR benchmark are shown in Figure 7.4 and tabulated with additional information in Table 7.3. Results for the IIR benchmark are shown in Figure 7.5 and tabulated with additional information in Table 7.4.

As expected, the StrongARM microprocessor has the worst performance among the architectures considered in this study, as it requires many instructions and execution cycles to execute a given kernel in a highly sequential manner. The lack of a single-cycle multiplier exacerbates this problem. Furthermore, each instruction is burdened by a great deal of energy overhead. All other architectures have more internal parallelism which allows them to have much better performance than the StrongARM processor. Pleiades and the TI processors can execute an FIR tap in a single cycle. Pleiades performs much better on the energy scale than the TI processors because the TI processors have a general-purpose design, incurring a great deal of energy overhead to each instruction. Pleiades, on the other hand, has the ability to create a hardware structures optimized for a given kernel and can execute operations with a relatively small energy overhead. Features such as zero-overhead looping reduce the instruction fetch overhead for the TI processors, but they still fall short of the performance achieved by Pleiades. The XC4003A executes 5 taps in a single cycle. The XC4003A is not very energy efficient, but it has the ability to use optimized shift-and-add multipliers, instead of the full multipliers used in the other architectures.

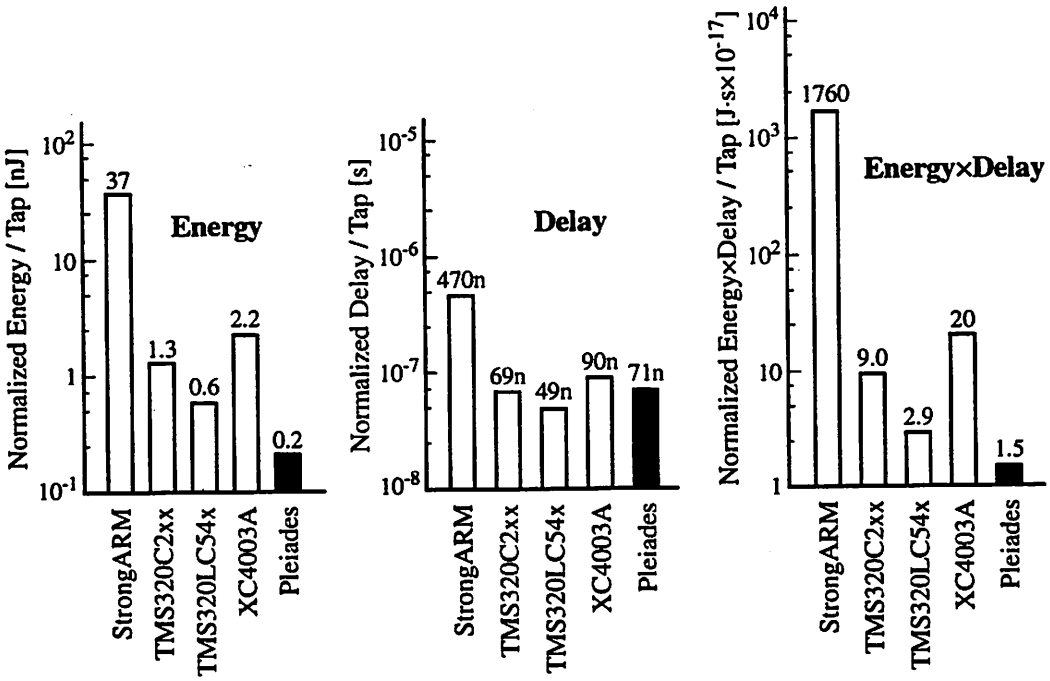


Figure 7.4: Comparison Results for the FIR Benchmark

Processor	StrongARM	TMS320C2xx	TMS320LC54x	XC4003A	Pleiades
Clock Frequency (MHz)	169	20	40	6	14
Number of Multipliers	0.5	1	1	5	1
Throughput (cycles/tap)	17	1	1	0.2	1
Energy/tap (nJ)	21.1	4.8	2.4	15.4	0.2
Capacitance/tap (pF)	8470	530	270	620	91
Norm. Capacitance/tap (pF)	16600	580	270	960	91
Norm. Energy/tap (nJ)	37.4	1.3	0.60	2.2	0.21
Norm. Delay/tap (ns)	470	69	49	90	71
Norm. Energy×Delay/tap ($J \cdot s \times 10^{-17}$)	1760	9.0	2.9	20	1.5

Table 7.3: Comparison Results for the FIR Benchmark

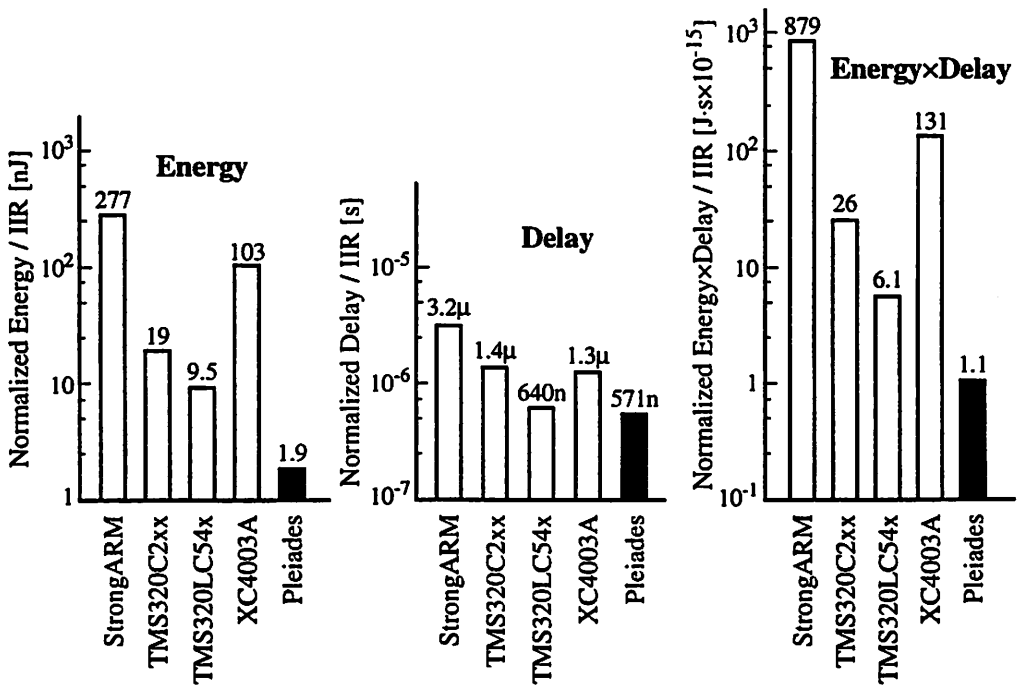


Figure 7.5: Comparison Results for the IIR Benchmark

Processor	StrongARM	TMS320C2xx	TMS320LC54x	XC4003A	Pleiades
Clock Frequency (MHz)	169	20	40	2.1	14
Number of Multipliers	0.5	1	1	9	2
Throughput (cycles/IIR)	114	20	13	1	8
Energy/IIR (nJ)	155	69	38	733	1.9
Capacitance/IIR (nF)	62.9	7.7	4.2	29.3	0.85
Norm. Capacitance/IIR (nF)	123	8.3	4.2	46	0.85
Norm. Energy/IIR (nJ)	277	18.7	9.5	103	1.9
Norm. Delay/IIR (ns)	3175	1370	640	1271	571
Norm. Energy×Delay/IIR ($J \cdot s \times 10^{-15}$)	879	25.6	6.1	131	1.1

Table 7.4: Comparison Results for the IIR Benchmark

Comparison results for the FFT benchmark are shown in Figure 7.6 and tabulated with additional information in Table 7.5. Compared to FIR and IIR benchmarks, the FFT benchmark is more complex. Pleiades outperforms the other processors by a large margin, owing to its ability to exploit higher levels of parallelism by creating an optimized parallel structure with minimal energy overhead.

7.1.7 Discussion

From the above results, we can see that the Pleiades architecture template, using the implementation style of the P1 prototype, achieves superior performance compared to other programmable architectures that are commonly used to implement signal processing algorithms. This superiority is attained in spite of the shortcomings of the P1 prototype that were detailed in Section 6.10.

7.2 Maia Results

The Maia processor was fabricated in a 0.25-μm CMOS technology [125, 126]. The chip contains 1.2 million transistors and measures 5.2×6.7 mm^2. It was packaged in a 210-pin PGA package. Die photo of Maia is shown in Figure 7.7. With a 1.0-V supply voltage, average throughput for kernels running on the satellite processors is 40 MHz. The ARM8 core runs at 40 MHz. The average power dissipation of the chip is 1.5 to 2.0 mW. Table 7.6 shows performance parameters of the various hardware components of the Maia processor.

Table 7.7 shows the energy profile of the VSELP speech coding algorithm, running on Maia. Six kernels were mapped onto the satellite processors. The rest of the algorithm is executed on the ARM8 control processor. The control processor is also responsible for configuring the satellite processors and the communication network. The

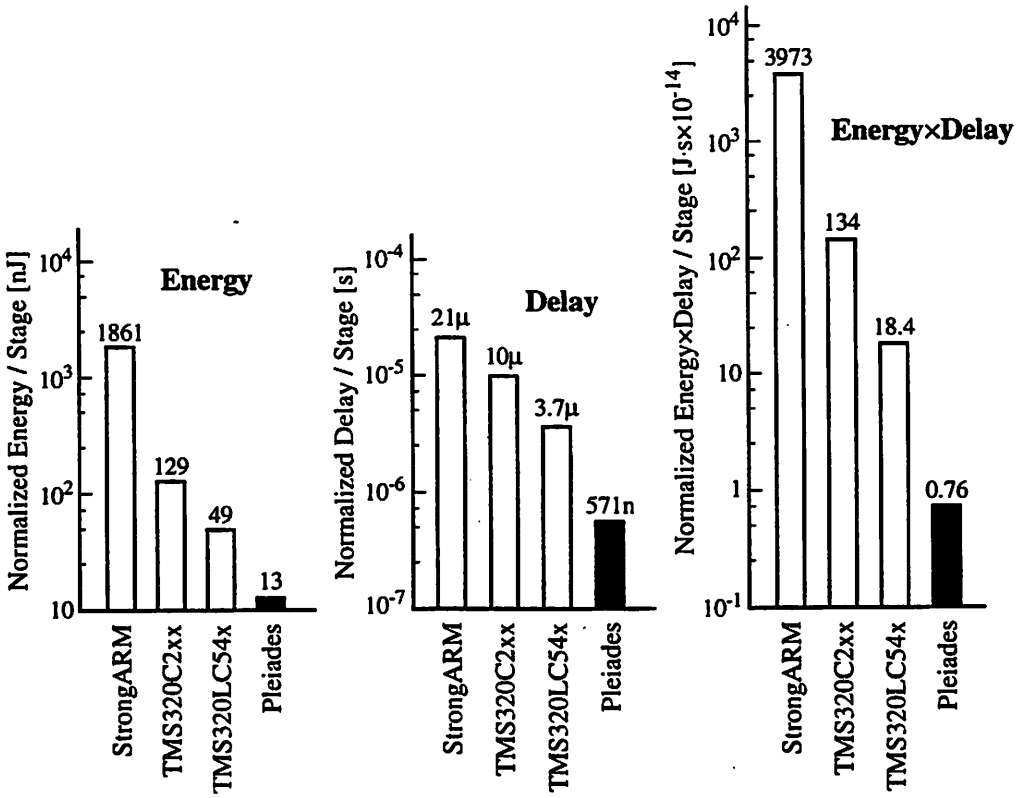


Figure 7.6: Comparison Results for the FFT Benchmark

Processor	StrongARM	TMS320C2xx	TMS320LC54x	Pleiades
Clock Frequency (MHz)	169	20	40	14
Number of Multipliers	0.5	1	1	4
Throughput (cycles/stage)	766	152	76	8
Energy/stage (nJ)	1040	478	197	13.3
Capacitance/stage (nF)	422	53.1	21.9	5.9
Norm. Capacitance/stage (nF)	827	57.3	21.9	5.9
Norm. Energy/stage (nJ)	1861	129	49.3	13.3
Norm. Delay/stage (ns)	21348	10412	3743	571
Norm. Energy×Delay/stage ($J \cdot s \times 10^{-14}$)	3973	134	18.4	0.76

Table 7.5: Comparison Results for the FFT Benchmark

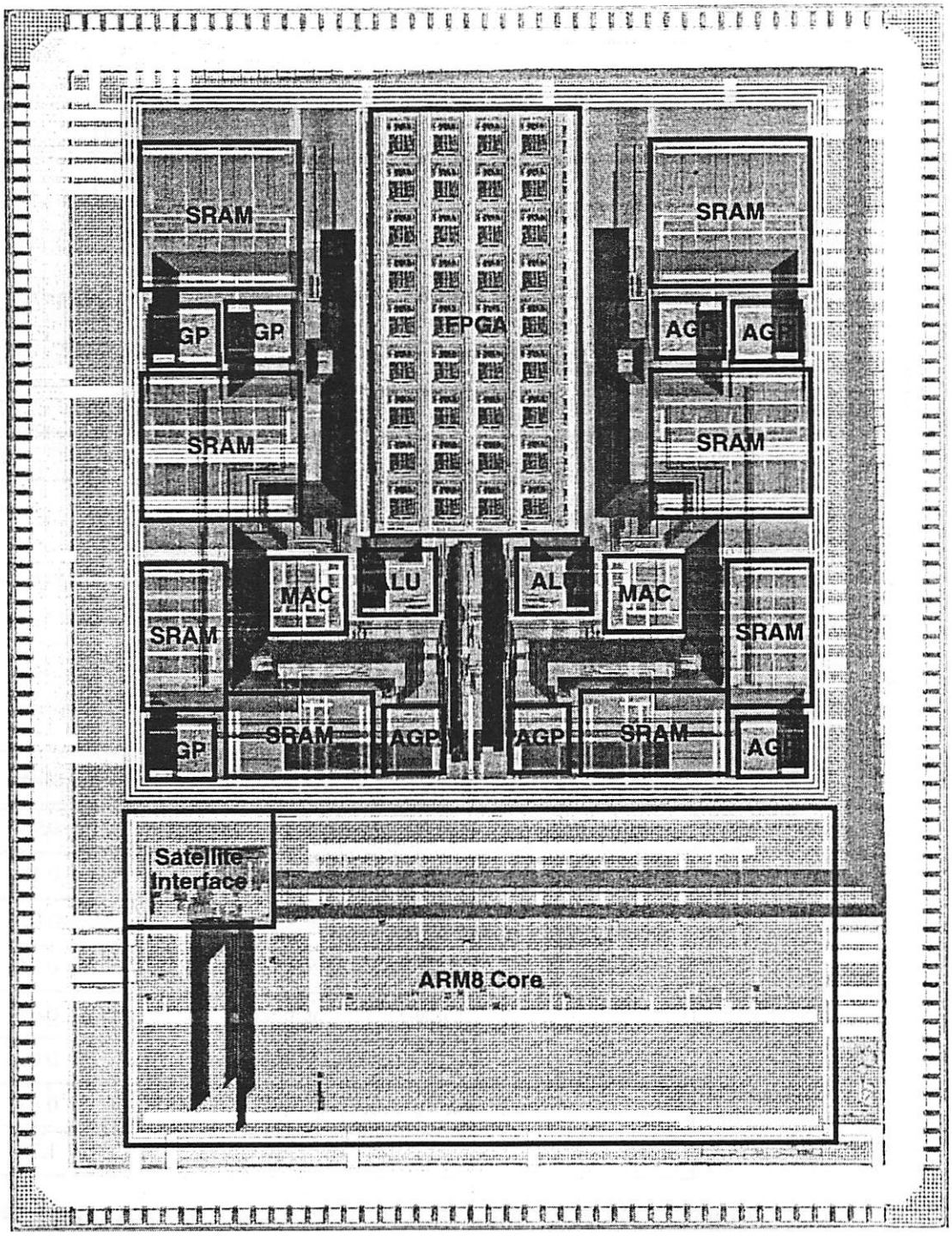


Figure 7.7: Die Photo of Maia

Component	Cycle Time (ns)	Energy per Cycle (pJ)	Area (mm^2)
MAC	24	21	0.25
ALU	20	8	0.09
SRAM (1K×16)	14	8	0.32
SRAM (512×16)	11	7	0.16
Address Generator	20	6	0.12
FPGA	25	18†	2.76
Interconnect Network	10	1‡	N/A

† This value is the average energy for various arithmetic functions.
‡ This value is the average energy per connection.

Table 7.6: Performance Data for Hardware Components of Maia

Function		Power (mW)
Kernels Running on Satellite Processors	Vector Dot Product	0.738
	FIR Filter	0.131
	IIR Filter	0.021
	Vector Sum with Scalar Multiply	0.042
	Code-Vector Computation	0.011
	Covariance Matrix Computation	0.006
Program Running on Control Processor		0.838
Total		1.787

Table 7.7: Energy Profile for the VSELP Algorithm Running on Maia

energy overhead of this configuration code running on the control processor is included in the energy consumption values of the kernels. In other words, the energy values listed in Table 7.7 for the kernels include contributions from the satellite processors as well as the control processor executing configuration code. The power dissipation of Maia when running VSELP is 1.8 mW. The lowest power dissipation reported in the literature to date is 17 mW for a programmable signal processor executing the Texas Instruments TMS320LC54x instruction set, implemented in a 0.25-μm CMOS process, running at 63 MHz with a 1.0-V supply voltage [149]. The energy efficiency of this reference processor is 270 μW/MHz, whereas the energy efficiency of Maia is 45 μW/MHz, which corresponds to an improvement by a factor of six.

CHAPTER 8

Conclusion

The problem addressed in this work was how to design a digital signal processor that is not only highly energy efficient, but it is also programmable and can be used to implement a variety of different, but similar, algorithms. The approach taken in this work was to explore ways of trading off flexibility for increased efficiency. This approach was based on the observation that for a given domain of signal processing algorithms, such as CELP-based speech coding, the underlying computational kernels that account for a large fraction of execution time and energy are very similar. What varies from algorithm to algorithm within a given domain are the parameters and the high-level control flow of those algorithms. By executing dominant kernels on dedicated, optimized processing elements that can execute those kernels with a minimum of energy overhead, significant energy savings can be achieved. Thus, the approach taken in this work yields processors that are domain-specific and are optimized for a given domain of algorithms. Thus, flexibility is traded off, allowing a designer to achieve high levels of energy efficiency, approaching that of a custom, application-specific design, while maintaining the flexibility needed to handle a variety of different algorithms within a domain of interest.

The main contribution of this work was a reusable architecture template, named Pleiades, that can be used to implement domain-specific, programmable processors for digital signal processing algorithms. The Pleiades architecture template relies on a heterogeneous network of processing elements, optimized for a given domain of algorithms, that can be reconfigured at run time to execute the dominant kernels of the given domain. Associated with the Pleiades architecture template is a design methodology. Defining this methodology was another contribution of this work. To explore and prove the effectiveness of the approach taken in this work, a prototype integrated circuit, named P1, incorporating all the elements of the Pleiades architecture template, was designed and fabricated in a 0.6-μm CMOS process. The P1 prototype and the subsequent benchmark study based on the results obtained from P1 provided early validation for the Pleiades approach. A number of important lessons were learned during the design and evaluation of P1. These lessons resulted in a number of important refinements to the Pleiades architecture template. Subsequent to P1, a domain-specific processor for CELP-based speech coding algorithms, named Maia, was designed. Maia was fabricated in a 0.25-μm CMOS process. It contains 1.2 million transistors and operates with a 1.0-Volt supply voltage. The energy efficiency achieved by Maia, in terms of power dissipation per computational throughput (Watt/MOPS), is six times higher than the best reference design reported in the literature.

8.1 Proposals for Future Research

The processor instances that were designed and implemented in this work focused on algorithm domains from baseband wireless applications. While the Pleiades architecture template is general in nature and can in principle be applied to other algorithm domains, it would still be worthwhile to explore other algorithm domains with different performance requirements and architectural parameters. One particularly important domain of algorithms is that of video coding algorithms that are based on the Discrete Cosine Transform (DCT). There is a variety of DCT-based algorithms and standards that

are widely used in video coding applications. The computational throughput required by these algorithms is very high, and it would be worth exploring the effectiveness of the Pleiades architecture template with these algorithms. Another important domain of algorithms is that of encryption/decryption algorithms, which are widely used in secure communications applications. These algorithms require processing elements with finer granularities than what is typically encountered in voice and video coding algorithms, and it would be worthwhile to explore the types of satellite processors that would be best suited for these algorithms.

Bibliography

[1] S. Sheng, A. Chandrakasan, and R. Brodersen, "A Portable Multimedia Terminal," *IEEE Communications Magazine*, pp. 64-75, December 1992.

[2] T. E. Truman, T. Pering, R. Doering, and R. W. Brodersen, "The Infopad Multimedia Terminal: A Portable Device for Wireless Information Access," *IEEE Transactions on Computers*, pp. 1073-1087, October 1998.

[3] K. Strehlo, "Advanced CMOS Technology Overtakes Established NMOS Applications," *Mini-Micro Systems*, pp. 115-123, July 1983.

[4] D. Bursky, "CMOS Microprocessors Outpace NMOS 8086/8088," *Electronic Design*, pp. 41-46, April 1984.

[5] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-Power Digital CMOS Design," *IEEE Journal of Solid-State Circuits*, pp. 473-484, April 1992.

[6] H. Veendrick, "Short-Circuit Dissipation of Static CMOS Circuitry and Its Impact on the Design of Buffer Circuits," *IEEE Journal of Solid-State Circuits*, pp. 468-473, August 1984.

[7] J. Rabaey, *Digital Integrated Circuits: A Design Perspective*, Prentice Hall, Upper Saddle River, New Jersey, 1996.

[8] A. Chandrakasan, *Low Power Digital CMOS Design*, Ph.D. Dissertation, University of California, Berkeley, 1994.

[9] A. Keshavarzi, K. Roy, and C. F. Hawkins, "Intrinsic Leakage in Low Power Deep Submicron CMOS ICs," *Proceedings of the International Test Conference*, pp. 146-155, November 1997.

[10] A. Chandrakasan and R. Brodersen, *Low-Power CMOS Design*, IEEE Press, Piscataway, New Jersey, 1998.

[11] T. D. Burd and R. W. Brodersen, "Energy Efficient CMOS Microprocessor Design," *Proceedings of the 28th Annual HICSS Conference*, pp. 288-297, January 1995.

[12] K. Usami and M. Horowitz, "Clustered Voltage Scaling Technique for Low-Power Design," *Proceedings of the 1995 International Symposium on Low Power Design*, pp. 3-8, April 1995.

[13] J. Chang and M. Pedram, "Energy Minimization Using Multiple Supply Voltages," *Proceedings of the 1996 International Symposium on Low Power Electronics and Design*, pp. 157-162, August 1996.

[14] L. S. Nielsen, C. Niessen, J. Sparso, K. van Berkel, "Low-Power Operation Using Self-Timed Circuits and Adaptive Scaling of the Supply Voltage," *IEEE Transactions on VLSI Systems*, pp. 391-397, December 1994.

[15] T. D. Burd and R. W. Brodersen, "Processor Design for Portable Systems," *Journal of VLSI Signal Processing*, pp. 203-221, August-September 1996.

[16] H. Zhang and J. Rabaey, "Low-Swing Interconnect Interface Circuits," *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pp. 161-166, August 1998.

[17] D. B. Lidsky and J. M. Rabaey, "Low-Power Design of Memory Intensive Functions Case Study: Vector Quantization," *Proceedings of the 1994 IEEE Workshop on VLSI Signal Processing*, pp. 378-387, October 1994.

[18] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, H. De Man, "Global Communication and Memory Optimizing Transformations for Low Power Signal Processing Systems," *Proceedings of the 1994 IEEE Workshop on VLSI Signal Processing*, pp. 178-187, October 1994.

[19] T. Sakuta, W. Lee, and P. T. Balsara, "Delay Balanced Multipliers for Low Power/ Low Voltage DSP Core," *Proceedings of the 1995 International Symposium on Low Power Electronics*, pp. 36-37, October 1995.

[20] A. Chandrakasan, R. Allmon, A. Stratakos, and R. W. Brodersen, "Design of Portable Systems," *Proceedings of the 1994 Custom Integrated Circuits Conference*, pp. 259-266, May 1994.

[21] M. Alidina, J. Monteiro, S. Devadas, A. Ghosh, and M. Papaefthymiou, "Precomputation-Based Sequential Logic Optimization for Low Power," *IEEE Transactions on VLSI Systems*, pp. 426-436, December 1994.

[22] P. E. Landman, *Low-Power Architectural Design Methodologies*, Ph.D. Dissertation, University of California, Berkeley, 1994.

[23] K. C. Pohlmann, *Principles of Digital Audio*, Third Edition, McGraw Hill, New York, 1995.

[24] C. E. Leiserson, F. M. Rose, J. B. Saxe, "Optimizing Synchronous Circuitry by Retiming," *Proceedings of Third Caltech Conference on Very Large Scale Integration*, pp. 87-116, March 1983.

[25] A. V. Oppenheim, R. W. Schafer, *Discrete-Time Signal Processing*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[26] G. D. Forney, Jr., "The Viterbi Algorithm," *Proceedings of the IEEE*, pp. 268-278, March 1973.

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

[28] A. Gersho and R. Gray, *Vector Quantization and Signal Compression*, Kluwer Academic Publishers, Boston, 1992.

[29] A. S. Spanias, "Speech Coding: A Tutorial Review," *Proceedings of the IEEE*, pp. 1541-1582, October 1994.

[30] L. R. Rabiner and R. W. Schafer, *Digital Processing of Speech Signals*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[31] J. C. Bellamy, *Digital Telephony*, Second Edition, Wiley, New York, 1991.

[32] N. S. Jayant and P. Noll, *Digital Coding of Waveforms: Principles and Applications to Speech and Video*, Prentice Hall, Englewood Cliffs, New Jersey, 1978.

[33] J. Makhoul, "Linear Prediction: A Tutorial Review," Proceedings of the IEEE, pp. 561-580, April 1975.

[34] J. D. Markel and A. H. Gray, Jr., Linear Prediction of Speech, Springer-Verlag, New York, 1976.

[35] M. R. Schroder and B. S. Atal, "Code-Excited Linear Prediction (CELP): High Quality Speech at Very Low Bit Rates," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pp. 937-940, 1985.

[36] P. Kroon and B. S. Atal, "Predictive Coding of Speech Using Analysis-by-Synthesis Techniques," Chapter 5 of *Advances in Speech Coding*, Kluwer Academic Publishers, Boston, Massachusetts, 1991.

[37] I. Gerson and M. Jasiuk, "Vector Sum Excited Linear Prediction (VSELP) Speech Coding at 8 kbps," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processning*, pp. 461-464, 1990.

[38] A. Yaqub and H. G. Moore, Elementary Linear Algebra with Applications, Addison Wesley, Reading, Massachusetts, 1980.

[39] EIA/TIA Interim Standard 54 (IS-54), EIA/TIA-PN2398, 1989.

[40] J. P. Campbell, Jr., V. C. Welch, and T. E. Tremain, "An Expandable Error-Protected 4800 bps CELP Coder," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pp. 735-738, 1989.

[41] J. Chen, "High-Quality 16 kb/s Speech Coding with a One-Way Delay Less than 2 ms," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pp. 453-456, 1990.

[42] S. Miki, K. Mano, T. Moriya, K. Oguchi, and H. Ohmuro, "A Pitch Synchronous Innovation CELP (PSI-CELP) Coder for 2-4 kbit/s," *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, vol. 2, pp. 113-116, 1994.

[43] R. Salami, C. Laflamme, J. Adoul, and D. Massaloux, "A Toll Quality 8 kb/s Speech Codec for the Personal Communication System (PCS)," *IEEE Transactions on Vehicular Technology*, pp. 808-816, August 1994.

[44] R. Salami *et al.*, "Design and Description of CS-ACELP: A Toll Quality 8 kb/s Speech Coder," *IEEE Transactions on Speech and Audio Processing*, pp. 116-130, March 1998.

[45] N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete Cosine Transform," *IEEE Transactions on Computers*, pp. 88-93, January 1974.

[46] T. Sikora, "MPEG Digital Video-Coding Standards," *IEEE Signal Processing Magazine*, pp. 82-100, September 1997.

[47] P. Pirsch, N. Demassieux, and W. Gehrke, "VLSI Architectures for Video Compression: A Survey," *Proceedings of the IEEE*, pp. 220-246, February 1995.

[48] M. J. Flynn, "Some Computer Organizations and Their Effeciveness," *IEEE Transactions on Computers*, pp. 948-960, September 1972.

[49] D. B. Skillicorn, "A Taxonomy for Computer Architectures," *IEEE Computer*, pp. 46-57, November 1988.

[50] A. DeHon, *Reconfigurable Architectures for General-Purpose Computing*, Ph.D. Dissertation, Massachusetts Institute of Technology, 1996.

[51] H. H. Goldstine, *The Computer: From Pascal to von Neumann*, Princeton University Press, Princeton, New Jersey, 1972.

[52] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Chapter 8, Morgan Kufmann Publishers, San Mateo, California, 1990.

[53] H. Mizuno and K. Ishibashi, "A Cost-Oriented Two-Port Unified Cache for Low-Power RISC Microprocessors," *1996 Symposium on VLSI Circuits Digest of Technical Papers*, pp. 72-73, 1996.

[54] S. Gary *et al.*, "PowerPC 603, A Microprocessor for Portable Computers," *IEEE Design & Test of Computers*, pp. 14-23, Winter 1994.

[55] S. Gary *et al.*, "The PowerPC 603 Microprocessor: A Low-Power Design for Portable Applications," *Proceedings of COMPCON*, pp. 307-315, 1994.

[56] S. Segars, K. Clarke, and L. Goudge, "Embedded Control Problems, Thumb, and the ARM7TDMI," *IEEE Micro*, pp. 22-30, October 1995.

[57] S. Segars, "ARM7TDMI Power Consumption," *IEEE Micro*, pp. 12-19, July/ August 1997.

[58] J. Montanaro *et al.*, "A 160MHz 32b 0.5W CMOS RISC Microprocessor," *International Solid-State Circuits Conference Digest of Technical Papers*, pp. 214-215, 1996.

[59] D. W. Dobberpuhl, "Circuits and Technology for Digital's StrongARM and ALPHA Microprocessors," *Proceedings of the Seventeenth Conference on Advanced Research in VLSI*, pp. 2-11, 1997.

[60] T. Nishitani, R. Maruta, Y. Kawakami, and H. Goto, "A Single-Chip Digital Signal Processor for Telecommunications Applications," *IEEE Journal of Solid-State Circuits*, pp. 372-376, August 1981 (Describes the NEC 7720).

[61] S. Magar, E. Caudel, and A. Leigh, "A Microcomputer with Digital Signal Processing Capability," *International Solid-State Circuits Conference Digest of Technical Papers*, pp. 32-33, 1982 (Describes the TI TMS32010).

[62] H. Kabuo et al., "An 80-MOPS-Peak High-Speed and Low-Power-Consumption 16-b Digital Signal Processor," *IEEE Journal of Solid-State Circuits*, pp. 494-503, April 1996.

[63] *TMS320C2X User's Guide*, SPRU014C, Texas Instruments, 1993.

[64] *TMS320C54X DSP Reference Set, Volume 1: CPU and Peripherals*, SPRU131F, Texas Instruments, 1999.

[65] *DSP1618 Digital Signal Processor Product Note*, Lucent, 1996.

[66] T. Shiraishi et al., "A 1.8V 36mW DSP for the Half-Rate Speech Codec," *Proceedings of the Custom Integrated Circuits Conference*, pp. 371-374, 1996.

[67] I. Verbauwhede et al., "A Low Power DSP Engine for Wireless Communications," *VLSI Signal Processing IX*, pp. 471-480, 1996.

[68] M. Hiraki *et al.*, "Stage-Skip Pipeline: A Low Power Processor Architecture Using a Decoded Instruction Buffer," *International Symposium on Low Power Electronics and Design Digest of Technical Papers*, pp. 353-358, 1996.

[69] J. Gray, A. Naylor, A. Abnous, N. Bagherzadeh, "VIPER: A VLIW Integer Miroprocessor," *IEEE Journal of Solid-State Circuits*, pp. 1377-1382, December 1993.

[70] A. Abnous, *Architectural Design and Analysis of a VLIW Integer Processor*, Masters Thesis, University of California, Irvine, 1991.

[71] *TMS320C6000 CPU and Insruction Set Reference Guide*, Texas Instruments, SPRU189D, 1999.

[72] `http://www.zsp.com/arch_arch.html`.

[73] P. M. Kogge, *Architecture of Pipelined Computers*, Hemisphere Publishing, New York, 1981.

[74] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw Hill, New York, 1984.

[75] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," *Proceedings of the International Computer Software and Applications Conference (COMPSAC)*, pp. 709-715, 1980.

[76] K. Aono *et al.*, "A Video Digital Signal Processor with a Vector-Pipeline Architecture," *IEEE Journal of Solid-State Circuits*, pp. 1886-1894, December 1992.

[77] J. Wawrzynek *et al.*, "Spert-II: A Vector Microprocessor System," *IEEE Computer*, pp. 79-86, March 1996.

[78] A. Preleg and U. Weiser, "MMX Technology Extension to the Intel Architecture," *IEEE Micro*, pp. 42-50, August 1996.

[79] R. Lee, "Real-Time MPEG Video via Software Decompression on a PA-RISC Processor," *Proceedings of IEEE COMPCON*, pp. 186-192, March 1995.

[80] M. Trembley, M. O'Conner, V. Narayanan, and L. He, "VIS Speeds New Media Processing," *IEEE Micro*, pp.10-20, August 1996.

[81] D. A. Carlson, R. W. Castelino, and R. O. Mueller, "Multimedia Extensions for a 550-MHz RISC Microprocessor," *IEEE Journal of Solid-State Circuits*, pp. 1618-1624, November 1997.

[82] U. Schmidt, K. Casesar, and T. Himmel, "Data-Driven Array Processor for Video Signal Processing," *IEEE Transactions on Consumer Electronics*, pp. 327-333, August 1990.

[83] H. Veendrick, O. Popp, G. Postuma, and M. Lecoutere, "A 1.5 GIPS Video Signal Processor (VSP)," *Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 95-98, 1994.

[84] A. K. Yeung and J. Rabaey, "A 2.4 GOPS Data-Driven Reconfigurable Multiprocessor IC for DSP," *International Solid-State Circuits Conference Digest of Technical Papers*, pp. 108-109, 1995.

[85] B. Ackland *et al.*, "A Single-Chip 1.6 Billion 16-b MAC/s Multiprocessor DSP," *Proceedings of the Custom Integrated Circuits Conference*, pp. 537-540, 1999.

[86] A. K. Yeung, *A Data-Driven Multiprocessor Architecture for High Throughput Digital Signal Processing*, Ph.D. Dissertation, University of California, Berkeley, 1995.

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

[88] A. DeHon, "Trends Toward Spatial Computing Architectures," *International Solid-State Circuits Conference Digest of Technical Papers*, pp. 362-363, 1999.

[89] J. E. Vuillemin *et al.*, "Programmable Active Memories: Reconfigurable Systems Come of Age," *IEEE Transactions on VLSI Systems*, pp 56-69, March 1996.

[90] E. Kusse, *Analysis and Circuit Design for Low Power Programmable Logic Modules*, Masters Thesis, University of California, Berkeley, 1997.

[91] C. A. R. Hoare, "Communicating Sequential Processes," Communications of the ACM, Vol. 21, No. 8, August 1978.

[92] P. A. Laplante, *Real-Time System Design and Analysis: An Engineer's Handbook*, Second Edition, IEEE Computer Society Press, New York, 1997.

[93] A. Varma and C. S. Raghavendra, *Interconnection Networks for Multiprocessors and Multicomputers: Theory and Practice*, IEEE Computer Society Press, New York, 1994.

[94] D. H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Transaction on Computers*, pp. 1145-1155, December 1975.

[95] H. Zhang, M. Wan, V. George, and J. Rabaey, "Interconnect Architecture Exploration for Low-Energy Reconfigurable Single-Chip DSPs," *Proceedings of the IEEE Computer Society Workshop on VLSI '99*, pp. 2-8, 1999.

[96] W. Tsu *et al.*, "HSRA: High-Speed Hierarchical Synchronous Reconfigurable Array," *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, pp. 125-134, 1999.

[97] H. Zhang and J. Rabaey, "Low-Swing Interconnect Interface Circuits," *Proceedings of the 1998 IEEE Symposium on Low-Power Electronics and Design*, pp. 161-166, 1998.

[98] H. Zhang, V. George, and J. Rabaey, "Low-Swing on-Chip Signaling Techniques: Effectiveness and Robustness," *IEEE Transactions on VLSI Systems*, pp. 264-272, June 2000.

[99] *XC4000E and XC4000X Series Field Programmable Gate Arrays*, Xilinx, Inc., 1999.

[100] S. Segars, "The ARM9 Family: High Performance Microprocessors for Embedded Applications," *Proceedings of the International Conference on Computer Design*, pp. 230-235, 1998.

[101] A. DeHon, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century," *Proceedings of the IEEE Workshop on FPGA Custom Computing Machines*, pp. 31-39, 1994.

[102] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A Time-Multiplexed FPGA," *Proceedings of the IEEE Workshop on FPGA Custom Computing Machines*, pp. 22-28, 1997.

[103] J. R. Hauser and J. Wawrzynek, "GARP: A MIPS Processor with a Reconfigurable Coprocessor," *Proceedings of the IEEE Workshop on FPGA Custom Computing Machines*, pp. 12-21, 1997.

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

[105] R. D. Fellman, "Design Issues and an Architecture for the Monolithic Implementation of a Parallel Digital Signal Processor," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, pp. 839-852, May 1990.

[106] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *IEEE Proceedings*, pp. 1235-1245, September 1987.

[107] E. A. Lee, "Consistency in Dataflow Graphs," *IEEE Transactions on Parallel and Distributed Systems*, pp. 223-235, April 1991.

[108] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, pp. 24-35, January 1987.

[109] P. Hoang and J. Rabaey, "A Compiler for Multiprocessor DSP Implementation," *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. V, pp. 581-584, 1992.

[110] Hoang and J. Rabaey, "Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput," *IEEE Trnasactions on Signal Processing*, pp. 2225-2235, June 1993.

[111] J. L. Pino, T. M. Parks, and E. A. Lee, "Automatic Code Generation for Heterogeneous Multiprocessors," *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. II, pp. 445-448, 1994.

[112] C. L. Seitz, "System Timing," in C. A. Mead and L. A. Conway, *Introduction to VLSI Systems*, Chapter 7, Addison-Wesley, Reading, Massachusetts, 1980.

[113] D. W. Dobberpuhl et al., "A 200-MHz 64-b Dual-Issue CMOS Microprocessor," *IEEE Journal of Solid State Circuits*, pp. 1555-1567, November 1992.

[114] K. van Berkel, R. Burgess, J. Kessels, M. Roncken, F. Schalij, and A. Peeters, "Asynchronous Circuits for Low Power: A DCC Error Corrector," *IEEE Design & Test of Computers*, pp. 22-32, Summer 1994.

[115] G. M. Jacobs and R. W. Brodersen. "A Fully Asynchronous Digital Signal Processor Using Self-Timed Circuits," *IEEE Journal of Solid State Circuits*, pp. 1526-1537, December 1990.

[116] T. E. Williams and M. A. Horowitz, "A Zero-Overhead Self-Timed 160-ns 54-b CMOS Divider, *IEEE Journal of Solid State Circuits*, pp. 1651-1661, November 1991.

[117] S. B. Furber and J. Liu, "Dynamic Logic in Four-Phase Micropipelines," Proceedings of the Second International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp. 11-16, 1996.

[118] S. B. Furber and P. Day, "Four-Phase Micropipeline Control Circuits," *IEEE Transactions on VLSI Systems*, pp. 247-253, June 1996.

[119] M. Benes, *Design and Implementation of Communication and Switching Techniques for the Pleiades Family of Processors*, Masters Thesis, University of California, Berkeley, 1999.

[120] V. Tiwari, S. Malik, A. Wolfe, and M. T. Lee, "Instruction Level Power Analysis and Optimization of Software," *Journal of VLSI Signal Processing*, pp. 223-238, August/September 1996.

[121] H. Zhang, M. Wan, V. George, and J. Rabaey, "Interconnect Architecture Exploration for Low-Energy Reconfigurable Signle-Chip DSPs," *Proceedings of the IEEE Computer Society Workshop on VLSI '99*, pp. 2-8, 1999.

[122] S.-F. Li, M. Wan, and J. Rabaey, "Configuration Code Generation and Optimizations for Heterogeneous Reconfigurable DSPs," *Proceedings of the 1999 IEEE Workshop on Signal Processing Systems*, pp. 169-180, October 1999.

[123] M. Wan et al., "A Low-Power Reconfigurable Dataflow Driven DSP System," *Proceedings of the 1999 IEEE Workshop on Signal Processing Systems*, pp. 191-200, October 1999.

[124] M. Wan, *A Design Methodology for Low-Power Heterogeneous Reconfigurable Digital Signal Processors*, Ph.D. Dissertation, University of California, Berkeley, 2001.

[125] H. Zhang, V. Prabhu, V. George, M. Wan, M. Benes, A. Abnous, and J. M. Rabaey, "A 1-V Heterogeneous Reconfigurable DSP IC for Wireless Baseband Digital Signal Processing," *IEEE Journal of Solid-State Circuits*, pp. 1697-1704, November 2000.

[126] H. Zhang, V. Prabhu, V. George, M. Wan, M. Benes, A. Abnous, and J. M. Rabaey, "A 1-V Heterogeneous Reconfigurable Processor IC for Baseband Wireless Applications," *International Solid-State Circuits Conference Digest of Technical Papers*, pp. 68-69, 2000.

[127] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen, "A Dynamic Voltage Scaled Microprocessor System," *IEEE Journal of Solid-State Circuits*, pp. 1571-1580, November 2000.

[128] V. Prabhu, *Integration of Embedded Processors in Wireless Systems-On-A-Chip*, Masters Thesis, University of California, Berkeley, 2000.

[129] V. George, H. Zhang, and J. Rabaey, "Low-Energy FPGA Design," *Proceedings of the International Symposium on Low-Power Electronics and Design*, pp. 188-193, 1999.

[130] V. George, *Low-Energy FPGA Design*, Ph.D. Dissertation, University of California, Berkeley, 2000.

[131] `http://www.mosis.org/.`

[132] A. D. Booth, "A Signed Binary Multiplication Technique," *Quarterly Journal of Mechanical and Applied Math*, pp. 236-240, 1951.

[133] O. L. MacSorley, "High Speed Arithmetic in Binary Computers," *Proceedings of IRE*, pp. 67-91, 1961.

[134] `http://www.synopsys.com/.`

[135] `http://www.cadence.com/.`

[136] `http://www.avanticorp.com/.`

[137] A. Abnous, K. Seno, Y. Ichikawa, M. Wan, and J. Rabaey, "Evaluation of a Low-Power Reconfigurable DSP Architecture," *Proceedings of the Reconfigurable Architectures Workshop*, pp. 55-60, 1998.

[138] *Digital Semiconductor SA-110 Microprocessor Technical Reference Manual*, Digital Equipment Corporation, 1996.

[139] *Digital Semiconductor SA-110 Microprocessor Evaluation Board Reference Manual*, Digital Equipment Corporation, 1996.

[140] `http://www.ti.com/`.

[141] *TMS320C5x General-Purpose Applications User's Guide*, Literature Number SPRU164, Texas Instruments, 1997.

[142] T. Anderson, *The TMS320C2xx Sum-of-Products Methodology*, Technical Application Report SPRA068, Texas Instruments, 1996.

[143] M. Tsai, *IIR Filter Design on the TMS320C54x DSP*, Technical Application Report SPRA079, Texas Instruments, 1996.

[144] `ftp://ftp.ti.com/pub/tms320bbs/c5xxfiles/54xffts.exe`, C'54x Software Support Files, Texas Instruments.

[145] C. Turner, *Calculation of TMS320LC54x Power Dissipation*, Technical Application Report SPRA164, Texas Instruments, 1997.

[146] C. Turner, *Calculation of TMS320C2xx Power Dissipation*, Technical Application Report SPRA088, Texas Instruments, 1996.

[147] T. C. Lee, V. Tiwari, A. Malik, and M. Fujita, "Power Analysis and Minimization Techniques for Embedded DSP Software," *IEEE Transactions on VLSI Systems*, pp. 123-135, March 1997.

[148] J. M. Rabaey at al., "Fast Prototyping of Data Path Intensive Architectures," *IEEE Design & Test Magazine*, pp. 40-51, June 1991.

[149] W. Lee et al., "A 1V DSP for Wireless Communications," *International Solid-State Circuits Conference Digest of Technical Papers*, pp. 92-93, 1997.