Dynamically Reconfigurable Architecture for a Class of Real-Time Applications

TakaHide Ohkami

TR92-03 December 1992

Abstract

This report (thesis) presents an architectural design methodology for computing systems suitable for a class of real-time applications, characterized by a large volume of periodic real-time data input at a high rate and vector operations on the real-time data. The proposed methodology incorporates into the architectural design the notion of resource sharing as well as techniques for satisfying timing requirements. The proposed design methodology is based upon a new computing system architecture called Dynamically Reconfigurable Architecture or DRA, which is suitable for the target class of real-time applications. Its most distinguished feature is in the dynamically reconfigurable computation network, which consists of arithmetic-operation-level functional modules interconnected through a switching network or multiple data buses that can be logically reorganized for pipelined vector computations for real-time data. The computation network is logically configured to form one or more arithmetic pipelines before vector operations are initiated and remains unchanged during the vector operations.

Mitsubishi Electric Research Laboratories

Technical Report 92-03 **January 1992**

Dynamically Reconfigurable Architecture for a Class of Real-Time Applications

by

TakaHide Ohkami

Abstract

This report (thesis) presents an architectural design methodology for computing systems suitable for a class of real-time applications, characterized by a large volume of periodic real-time data input at a high rate and vector operations on the real-time data. The proposed methodology incorporates into the architectural design the notion of resource sharing as well as techniques for satisfying timing requirements.

The proposed design methodology is based upon a new computing system architecture called *Dynamically Reconfigurable Architecture* or *DRA*, which is suitable for the target class of real-time applications. Its most distinguished feature is in the *dynamically reconfigurable computation network*, which consists of arithmetic-operation-level functional modules interconnected through a switching network or multiple data buses that can be logically reorganized for pipelined vector computations for real-time data. The computation network is logically configured to form one or more arithmetic pipelines before vector operations are initiated and remains unchanged during the vector operations.

Submitted in partial fulfillment of the requirement for the degree of Doctor of Philosophy at the University of Tokyo.

201 Broadway
Cambridge Massachusetts 02139

Publication History:-

1. First printing, TR 92-03, January 1992

Contents

Chapter 1

Introduction

This chapter is an introduction, giving the background and motivation of the research work. It also presents the organization of this thesis, which reflects the way a real-time computing system should be designed.

Real-Time Computing

Today, real-time applications are ubiquitous in our society. These applications include factory automation, process control, flight control, undersea exploration, space exploration, robotics, medical image processing, transaction processing, telecommunications, fire control, missile guidance, and so on. They are different from other applications in that timing requirements are a major factor in their tasks, since real-time tasks depend not only on value but also on time. In other words, real-time tasks are required to produce correct results in a timely manner. Some applications may cause serious damage if the required timing constraints are missed. Timing requirements vary greatly depending on applications, while some applications may specify the precise execution timing for each task, others may specify only the average execution timing for a collection of tasks.

Signal processing is one of the large real-time application areas. It includes applications in such fields as speech, image, sonar, radar, robotics, and telecommunications, to name a few. In a typical real-time signal processing application, real-time signals are collected periodically and processed, probably through a variety of transformations (e.g., Fast Fourier Transform or FFT). Some useful information is extracted from the original input signals and then used to drive the other parts of the application system.

Transaction processing is another large area of real-time applications. It is quite different in nature from signal processing and very popular in business environments, where business data processing systems have been used on-line for more than two decades. Airline reservation systems and banking systems are well-known examples. Such systems have a database, centralized or distributed, and perform update and retrieval operations on the database upon request of users. In transaction processing applications, the average processing time, which is what a user observes directly, is a major factor of design.

Although real-time applications have been prevailing in our society, real-time computing itself has not been recognized as a scientific field. No firm scientific base has been established yet to handle timing requirements in a systematic manner in real-time computing. Because of this lack of the scientific base, many real-time systems have been constructed in an ad hoc manner, and most of them have been tailored to specific applications, showing inflexibility for the other type of applications. This design tradition has not contributed much to the accumulation of scientific knowledge on real-time computing, but has produced many misconceptions about it, which in turn reflect the current status of the field. J. A. Stankovic has pointed out some of the prevalent misconceptions in [170], [171]:

(1) There is no science in real-time system design.
(2) Advances in supercomputer hardware will take care of real-time requirements.
(3) Real-time computing is equivalent to fast computing.
(4) Real-time programming is assembly coding, priority interrupt programming, and writing device drivers.
(5) Real-time systems research is performance engineering.
(6) The problems in real-time system design have all been solved in other areas of computer science or operations research.
(7) It is not meaningful to talk about guaranteeing real-time performance because we cannot guarantee that the hardware will not fail and the software is bug-free or that the actual operating conditions will not violate the specified design limits.
(8) Real-time systems function in a static environment.

Stankovic argues against these widespread misconceptions and identifies research issues to establish a scientific base in the area of real-time computing in his papers [170], [171].

Advances in VLSI and Computing Technologies

Since the first integrated circuits appeared three decades ago, the circuit density (the number of transistors fabricated on a unit area) has been dramatically increased on semiconductor chips. J. D. Meindl expects the progress of the semiconductor technologies to continue in the 1990s, leading to the Giga-Scale Integration or GSI, which contains one billion transistors on a chip [125]. Improvements have been, are, and will be, made in switching speed as well as in circuit density, especially on silicon chips.

Advances in VLSI technologies have been a major thrust for research and development of advanced computing systems. Computing technologies have been improved to exploit the improvements in semiconductor technologies. The increase in switching speed of semiconductor devices has contributed directly to the design of high-performance computing systems, which in turn has made feasible many time-consuming computing tasks like large-scale simulations. The increase in circuit density of semiconductor devices has dramatically reduced the size of computing systems. The miniaturization of electronic components has reduced the cost of computing systems to a great extent, turning computers into intellectual commodities. A. Peled predicts that computers will become an intellectual utility, widely available, ultimately as ubiquitous as the telephone set [150]. The decrease in the cost of computing systems has made it possible to design a large high-performance computing system that uses a large number of identical smaller computing elements.

Conventional computing systems have been designed to perform operations in sequence, that is, one operation at a time. With the increasing availability of cheap computing elements, such as microprocessors, it has become feasible to design a computing system that contains a number of small computing elements, each having a capability of performing operations in sequence. Parallel computing is potentially much more powerful and much faster than sequential computing. Since the computing speed of a single processor is bounded by the speed of electricity, which is in turn bounded by the speed of light ($c = 2.99792 \times 10^8$ m/sec in a vacuum), the fastest speed at which signals can travel, we resort to multiple processors if we go beyond that speed.

As discussed by G. C. Fox and P. C. Messina in [63], parallel computing machines come in various flavors with respect to computing style, processor connection, memory access, etc. However, no single parallel architecture can cover all technical problems. We will review some of them in Chapter 3. In order to use as much computing power as possible for solving a problem, we need to find the best match between the parallel architecture and the problem to solve. One of the most important engineering problems is the design of the parallel architecture that best fits the class of problems to solve. There is no well-known general guiding principle for such a design problem; however, we have accumulated some knowledge of the matching pairs of architecture and problems. For example, it is well-known that the architecture with mesh-connected processor elements matches easily the problem of image processing.

Parallel hardware produces the raw (parallel) computing power to be used for solving a problem, which is coded as an application program. Between the hardware and the application program lies the parallel system software (operating systems, run-time libraries, compilers, etc.) which tries to exploit the full capabilities of the hardware and make them available to application programs with as little computing power as possible. This process must be efficient, since the computing power consumed by it is observed as a system overhead when an application program executes. In general, however, it is not an easy task to design efficient system software. Many difficult problems arise in parallel programming, particularly in the areas of coordinating interrelated subtasks, constituting a single task, which are performed on different processors running in parallel. D. Gelernter discusses these topics in [69].

For the other aspects of computing technologies, see [98] for storage technologies, [59] for user-computer interface technologies, and [91] for computer networking technologies.

Research Overview

Computing technologies have been advancing along with VLSI technologies; however, real-time computing has not reaped large benefits from the advances in computing and VLSI technologies. This is partly because the scientific base in the field is too weak to absorb the advances of the other fields.

Given an opportunity for research and development of a large-scale real-time computing system for radar signal processing, we were well motivated to contribute to building a scientific base for that field by developing a design methodology for real-time computing systems.

A review of conventional real-time computing systems for signal processing reveals the ad hoc design in their architecture; design features are dedicated to particular aspects of the applications. These design features are individual and not always coordinated from the viewpoint of system design. The notion of resource sharing is missing in the system design. This notion is important not only for design efficiency but also for design flexibility. Resource sharing is a fundamental concept that has been incorporated into the general computing system architecture. However, many conventional real-time computing systems, particularly for signal processing, have been designed in an ad hoc manner as a collection of features, missing the resource sharing concept.

With the increasing availability of inexpensive microprocessors, many microprocessor-based systems have been developed for small- and medium-scale signal processing applications that do not require high performance. These systems are more flexibile than previous systems without a microprocessor, since flexibility is obtained by programming a microprocessor. While resource sharing is realized around the microprocessor to some extent, ad hoc design techniques are still used for large-scale signal processing applications. We were interested in incorporating the notion of resource sharing into the system design for high-performance large-scale real-time signal processing applications.

A review of well-known computer architectures reveals that the current computer architectures are not quite suitable for real-time signal processing applications that handle a large quantity of periodic signals. The results of this review will be presented in Chapter 3.

Through an exploration of the new architecture that best fits the applications while maintaining the architectural generality and flexibility. several key features from the well-known computer architectures can be combined and incorporated into a new architecture, and used as a base architecture for a class of real-time applications. This will be fully discussed in Chapter 4.

The major feature of the proposed architecture is a computation network that can be dynamically reconfigured for different pipelined vector computations by changing logical interconnections of arithmetic-operation-level functional modules. The computation network dynamically forms one or more arithmetic pipelines for vector processing. It allows computational resources to be shared among different vector computations. The timing requirements are easily satisfied by forming the arithmetic pipelines in the computation network, since pipeline processing is exact and predictable in terms of timing. Another feature is the system structure for system-level pipeline processing that enables a large amount of real-time data to flow through the system to satisfy periodic timing requirements.

Thesis Organization

The chapters of this thesis can be grouped into the following parts:

- Part 1: The Scope of Work (Chapters 1–3)
- Part 2: The Proposed Architecture (Chapters 4–6)
- Part 3: Discussions and Conclusions (Chapters 7–8)

Part 1 (The Scope of Work) presents the scope of the research work. This part first overviews general trends in real-time computing, VLSI technologies, and computing technologies. It will describe the characteristics of real-time applications, identify the problems in architectural design of real-time systems, and define the target class of applications for which a new architecture is proposed. A review of the existing computer architectures and their viability for the target class of applications is also contained in this part.

Part 2 (The Proposed Architecture) proposes a new computing system architecture for the target class of real-time applications, since none of the existing architectures are suitable for the target class of real-time applications. This part will first present the design concepts of the real-time system architecture required for the target class of real-time applications, and will present theoretical models for performance and execution. A proposal of new computing system architecture, called the *Dynamically Reconfigurable Architecture* or *DRA*, is presented, along with its architectural features and the programming techniques for the proposed architecture. It also describes an example implementation of the proposed architecture, developed for real-time signal processing.

Part 3 (Discussions and Conclusions) gives discussions and concluding remarks.

The rest of this thesis is organized as follows.

Chapter 2 first identifies the characteristics of real-time applications, and goes on to describe the target class of real-time applications, at which the proposed architecture is targeted. Chapter 3 reviews the major existing computing system architectures (dedicated, pipeline, parallel, massively parallel, systolic array, VLIW, and other architectures) and examines their advantages and disadvantages for the target class of real-time applications.

Chapter 4 describes the design concepts for the design of a real-time computing system architecture for the target class of real-time applications. It also presents two theoretical models of performance and execution. The performance models include the vector and pipeline performance models. The execution model is based on a logical machine model and pipelined vector operations. Chapter 5 describes the general system structure of the proposed architecture and its major components with emphasis on the architectural features for real-time computing. It also describes the computation network, which consists of functional modules interconnected via a routing network, and the mechanisms to reorganize the computation network. Programming techniques for the proposed architecture are also covered. Chapter 6 gives an example implementation of the proposed architecture, which was designed primarily for high-speed real-time signal processing applications that handle a large volume of periodic radar signals.

Chapter 7 presents discussions, and Chapter 8 gives concluding remarks.

Bibliographic Notes gives brief notes on the referenced literature, and Bibliography lists all the references.

Chapter 2

Real-Time Computing

This chapter identifies the characteristics of real-time computing and the technical issues to be addressed. It also characterizes the class of real-time computing at which the proposed architecture is targeted.

2.1 Characteristics of Real-Time Computing

Real-time computing systems have been designed and used in many application areas as mentioned in the previous chapter. They differ from other computing systems in that timing requirements are a major factor of design. Timing faults of a real-time system may cause catastrophic consequences in some applications. The design of a real-time computing system must be not only logically correct, but also meet the imposed timing constraints. The correctness of the design is required for any type of computing systems; the timing constraints for the design are unique for real-time computing systems.

S.-C. Cheng, J. A. Stankovic, and K. Ramamritham classified real-time systems into two types in [37]: *soft* and *hard* real-time systems. According to their definition, in soft real-time systems tasks are performed by the system as fast as possible, but they are not constrained to finish by specific times; in hard real-time systems tasks have to be performed in a timely fashion. When we schedule real-time tasks, we need to find a schedule such that the total execution time of the tasks is minimum for a soft real-time system; we need to find a schedule such that each task meets its deadline and the total execution time is minimum for hard real-time systems. Scheduling algorithms for hard real-time tasks are surveyed in [37].

There are a large variety of timing requirements. However, they are either *periodic* or *aperiodic*. Periodic timing requirements arise typically from the periodic nature of the physical environment. For example, a system with a number of sensors that sample data at certain intervals imposes periodic timing requirements. Aperiodic timing requirements involve the deadline by which tasks start and/or finish. For instance, a system that reacts to the increase of the temperature beyond a certain level by starting a task within the specified time imposes aperiodic timing requirements.

From another point of view, timing requirements can be classified into *precise* and *statistical* categories. Precise timing requirements are imposed on individual tasks, while statistical timing requirements on a collection of tasks. In applications with precise requirements, each task must satisfy the specified timing requirements, which may be periodic or aperiodic. On the other hand, in the applications with statistical requirements, each task need not satisfy the specified timing requirements, but the collection of tasks must meet the requirements statistically. An application, for example, may require that 90% of tasks should be performed within, say, five seconds.

Transaction processing applications impose statistical timing requirements, and their design is based upon the statistical requirements. Since a typical transaction processing system involves many components with probabilistic behavior, such as disk I/O operations, multiprogramming, virtual memory, caching, network communications, etc., the behavior of the applications running on it is affected by the probabilistic system components. Moreover, transaction processing systems have to maintain the transaction properties for a collection of transactions, which make the systems more statistical. The processing time of each transaction can vary, but the collection of transactions is expected to meet the timing requirements. The response time is a typical measure to evaluate an on-line transaction processing system, which takes a request from a user, accesses to the database, and returns a message to the user. A. Leff and C. Pu discussed the problems in building a transaction processing system and tried to classify transaction processing systems based on the abstracted transaction properties in [116]. Performance modeling of transaction processing systems is described by W. H. Highleyman in [76]. T. Matsuzawa, N. Ogawa, T. Ohkami, and T. Noji presented a tool, called SMART, to predict the OLTP (On-Line Transaction Processing) performance [123], which was built based on the results from the performance analysis of OLTP systems done by N. Ogawa, Y. Yamanaga, T. Matsuzawa, T. Ohkami, and T. Noji [132].

Human interface can be also considered to be a real-time system. It doesn't impose the timing requirements that are specified numerically, but it imposes the fuzzy psychological timing requirements. It should make the user feel comfortable with its reactions to the user's requests. While the experienced user wants a quick reaction to his/her request, a novice may not want such a quick reaction. Some human interface may be required to process the user's request in the time within the specified range. For example, the human interface of some on-line tutorial may be required to change pages on screen in, say, no more than 10 seconds and no less than 6 seconds.

Reliability issues are closely related to real-time computing applications. Many real-time systems are required to continue to operate in the presence of minor system faults, since they are unattended or inaccessible after their installation, or critical to the entire system like a life support system, a nuclear power plant, or a spaceship. Although various techniques have been developed in the area of fault-tolerant computing, real-time computing has raised new problems. A fault-tolerant scheduling problem is one of such problems. The problem is to find an efficient way to schedule tasks to meet the timing constraints in the presence of a failure. This problem has been addressed, for instance, by A. L. Liestman and R. H. Campbell in [120] and C. M. Krishna and K. G. Shin in [97]. A. L. Liestman and R. H. Campbell proposed a deadline mechanism that guarantees that a primary algorithm meets a deadline in the presence of no failure and an alternative algorithm of less precision meets it in the presence of failure [120]. C. M. Krishna and K. G. Shin presented a dynamic programming algorithm that embeds backup (or contingency) schedules within the primary schedule to ensure that deadlines can be met in case of up to a certain number of processor failures [97]. Note that transaction processing systems have been developed with transaction recovery techniques from their beginning.

2.2 Architectural Design Problems

The major problem in the conventional real-time system architectures is the ad hockery in design. Many of them are based on the hardware and software dedicated to the particular aspects of the application that runs on them. This is in part due to the fact that no widely accepted general system design methodology for real-time computing systems is available, and in part due to the fact that once the requirements from the application are identified, complexity is less in the design of the architecture dedicated to the application than in the design with a more general architecture, since its major work is to collect the *feature modules*, which may be hardware, software, or both, each dedicated to a particular aspect of the applications, without considering much about the interactions between the feature modules. The design ad hockery of the existing real-time system architectures can be considered to reduce the design complexity by identifying the application requirements, grouping them into the aspects independent of each other, building the feature modules, each dedicated to an aspect, and connecting them together to form a single system.

The conventional design style is thus based on the collection of the feature modules, which are independent of each other. We mean by “independent” that an operation associated with an aspect of the application is performed in one and only one feature module. Although the feature modules are independent of each other, they are not necessarily prime to each other, that is, they may have many functions in common. This is illustrated in Figure 2.1, where the application X consists of the four aspects $\{A_1, A_2, A_3, A_4\}$, each consisting of operations in $\{a, b, c, d, e, f, g, h\}$, and the architecture Y consists of the feature modules $\{F_1, F_2, F_3, F_4\}$, each F_i corresponding to A_i and consisting of functions in $\{a, b, c, d, e, f, g, h\}$. Each F_i is independent of each other, since it corresponds to A_i and it is the one and only one feature module used when A_i is executed. However, as seen in Figure 2.1, some functions

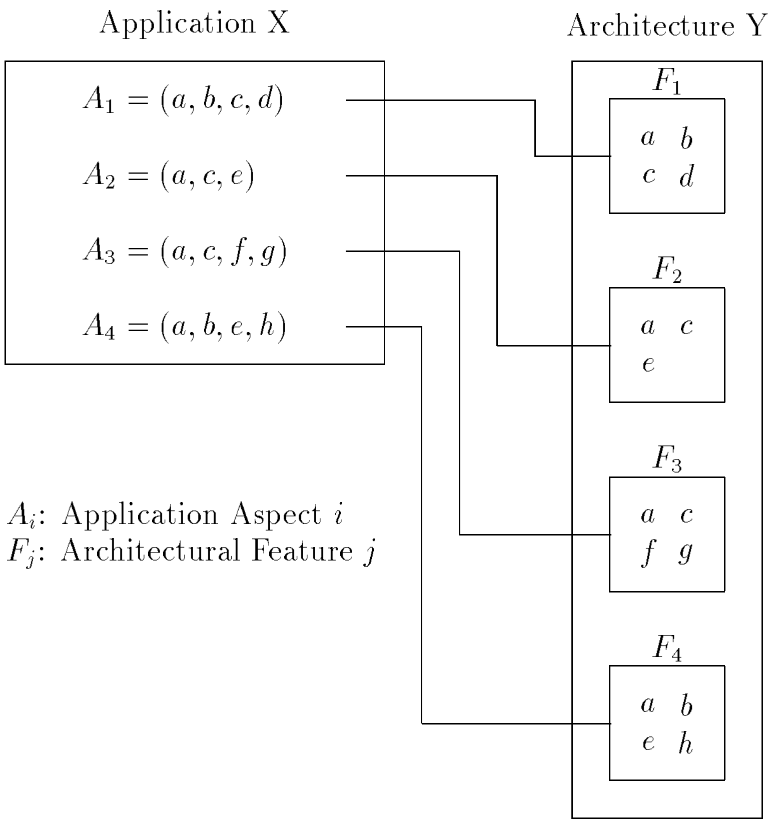


Figure 2.1: Architecture with Non-Prime Feature Modules.

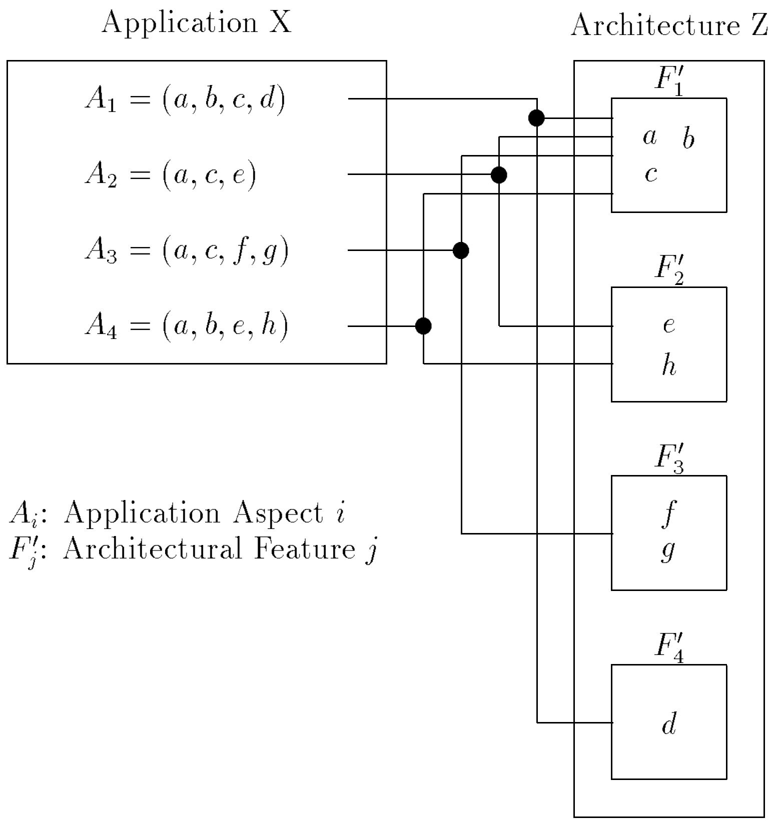


Figure 2.2: Architecture with Prime Feature Modules.

like a and b are included in more than one feature module. This redundancy of functions is not intended for fault tolerance, but just comes directly from the aspects of the application. As long as the total volume of the architecture is reasonable, this redundancy does not limit the design. But it becomes a problem when the total volume of the architecture exceeds a certain level.

When we construct a large-scale real-time computing system, this redundancy limits the reasonable implementation of the system. Consider, for example, a function with $f(n)$ components for the number of inputs n. When we scale up the function for the number of inputs $k \cdot n$, the required components for it is $f(kn)$. The total number of components required for all the duplicated functions to implement the scaled-up system becomes $f(kn) \cdot \alpha$, where α is the number of duplications. For instance, if $f(n) = n^2$, then the total number of components required for the scaled-up system is $\alpha \cdot k^2 n^2$. Since the number of components in a function with n inputs is equal to or greater than $O(n)$ in many cases, the scaling up of the system causes the super-linear increase of the component count, implying that the larger the system becomes the harder its implementation becomes with respect to volume.

Thus we need to incorporate the notion of *resource sharing* into the design of a real-time computing system. It is a very simple concept and not a new concept at all in computing technologies. It has been a common practice to use a single system component repeatedly as many times as possible. The ideal architecture for the application shown in Figure 2.1 is the architecture Z shown in Figure 2.2, where no two feature modules include the same function, that is, the feature modules are prime to each other. We call it the *prime-feature architecture*, while we call the previous type of architecture the *non-prime-feature architecture*. We call the design process of the prime-feature architecture *architectural prime factoring*, since it is analogous to prime factoring in mathematics.

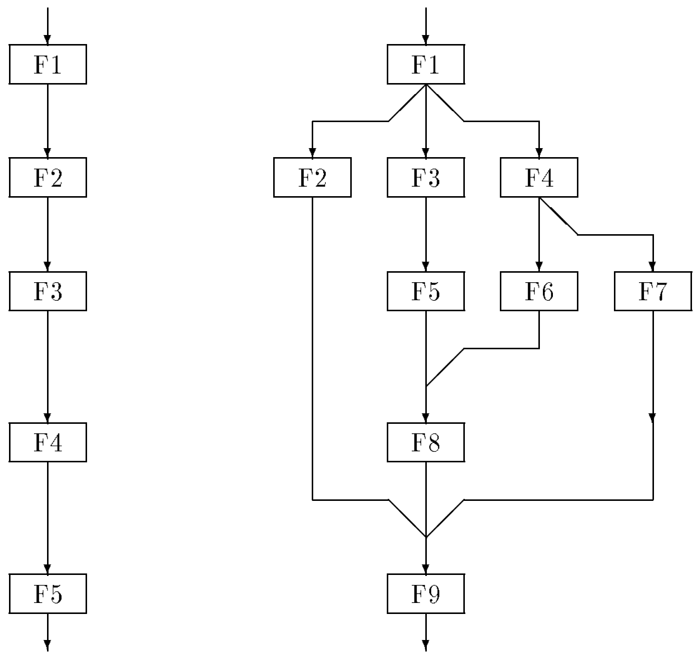
The architectural prime factoring consists of identifying the functions that perform the operations required for each aspect of an application and grouping them into prime feature modules. The notion of resource sharing can be automatically incorporated into the design in the process of architectural prime factoring. This is conceptually simple, but not necessarily simple in practice.

There is a timing problem in architectural prime factoring for the design of a real-time computing system architecture. In the non-prime-feature architecture, when an aspect of the application is executed, the function associated with the aspect performs all the operations required for the aspect. Therefore, the timings of the aspect can be checked at that function. That is, we need to pay attention to a single time flow through the function associated with the aspect. In the prime-feature architecture, the required operations are distributed to multiple functions, and the timings of the aspect must be checked at all the functions performing the operations. That is, we need to pay attention to a time flow split over the functions associated with the aspect. It increases the design complexity, especially for the timing-sensitive applications. Figure 2.3 illustrates the two cases.

While resource sharing reduces the total volume of architecture, it increases the interactions between the feature modules. No interaction between the feature modules is free for charge; any interaction induces a timing overhead, which is an extra charge due to resource sharing. One of such interactions in hardware is data transfer from one module to another. If the system handles a large number of data, data transfer may increase the amount of hardware or delay the availability of data for the subsequent modules. Thus the design complexity is increased by the interactions between the feature modules.

2.3 The Target Class of Real-Time Applications

We can easily construct a small-scale system around a microprocessor these days, since powerful microprocessors are now inexpensive and generally available. However, it is not an



(a) A Single Time Flow. (b) A Split Time Flow.

Figure 2.3: Time Flows Through Modules.

easy job to construct a large-scale system without scientific disciplines. In turn, scientific knowledge and experience are accumulated by building a large-scale system, since some systematic work is required to complete the construction.

We are interested in a large-scale real-time computing system for the real-time applications with periodic timing constraints, which periodically take, process, and produce a large amount of data. The system architecture that handles periodic data depends on the following key properties of the applications:

- the amount of periodic input data
- the rate of input data
- the types of operations on data

These properties direct the basic structures of the system. The amount of periodic input data dictates its data transfer structure. The rate of input data affects its control structure. The types of operations on data influence its functional structure.

We had an opportunity to design a large-scale real-time computing system for a radar signal processing application. It belongs to a class of the applications defined in terms of the above-mentioned properties. The applications in that class have in common the following properties:

- the amount of periodic input data: large
- the rate of input data: high
- the types of operations on data: vector operations

The amount of periodic input data is so large that a processor cannot process them online without buffering them in a memory. The rate of input data is so high that a simple scalar processor cannot handle data. The types of operations on data are vector operations; the same operations are repeatedly performed on all the data items in a group. Figure 2.4 shows a typical timing chart for processing such data, where data input, data process, and data output are overlapped, because input and output data are buffered before and after data process, respectively.

Figure 2.5 shows a signal processing system for a phased-array radar, which consists of a number of active antenna elements, collectively forming a single radar antenna. While traditional radars scan mechanically, a phased-array radar scans electronically by controlling the phases of power fed to antenna elements [128]. A variety of signal processing algorithms are used to convert radar signals obtained at the antenna elements into radar images [29]. When a radar system operates in a search mode or a tracking mode, its output is small in volume. However, when it operates in an image mode, its output is a large number of image pixels.

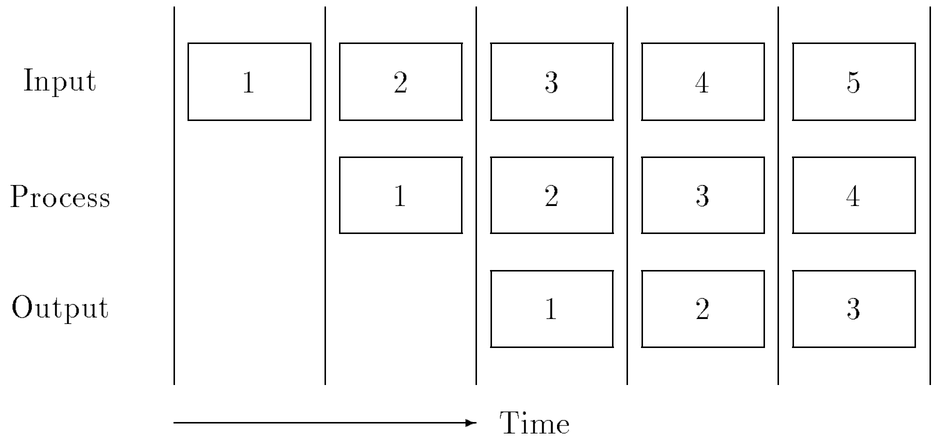


Figure 2.4: Periodic Real-Time Data Processing.

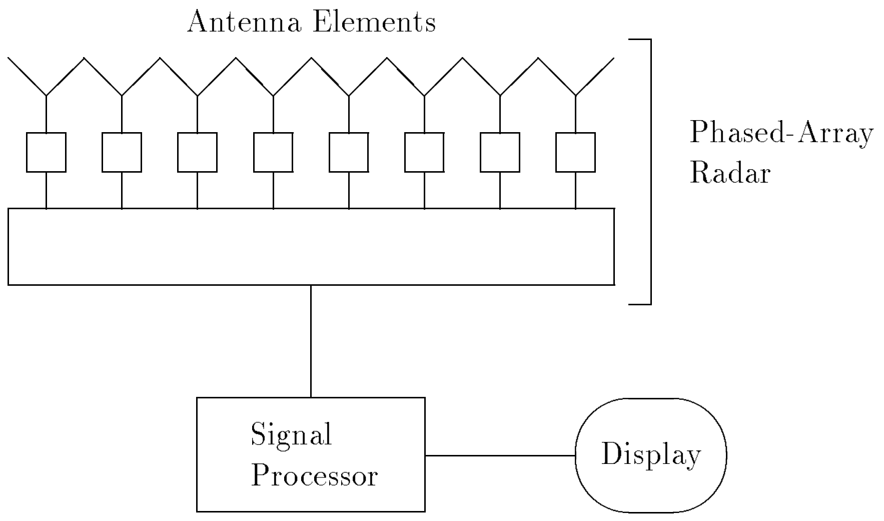


Figure 2.5: A Phased-Array Radar System.

Chapter 3

Review of Computer Architectures

This chapter reviews the following major computer architectures:

- Dedicated Architecture
- Pipeline Architecture
- Parallel Architecture
- Massively Parallel Architecture
- Systolic Architecture
- VLIW Architecture
- Hypercube Architecture
- Data-Flow Architecture
- RISC Architecture
- Superscalar Architecture

They are evaluated for the target class of real-time applications.

3.1 Dedicated Architecture

The dedicated architecture is the architecture designed for specific applications. Many architectural features are dedicated to particular aspects of the applications. We see many examples in the area of signal processing.

Fourier Transform [23] has been (and still is) of great importance in signal processing. In the history of signal processing, the invention of Fast Fourier Transform or FFT by J. W. Cooley and J. W. Tukey in 1965 [42] triggered the subsequent development of a variety of digital signal processing techniques in the 1960s and 1970s, along with the development of digital computers. Many digital signal processing algorithms, including convolution, correlation, and digital filters as well as FFT, were developed on digital computers. FFT has many variations in the way of computation [24], but it is basically an algorithm to compute discrete Fourier Transform or DFT in $O(n \log n)$, much faster than the straightforward DFT algorithm that takes $O(n^2)$. See [2, Chapter 7, pp.251–276], for example, for the computational complexity of FFT. The FFT algorithm contributed much to the development of real-time signal processing systems involving Fourier Transform. Since Fourier Transform has been a very time-consuming computation, compared with the other transformations, a variety of hardware implementations have been developed [20], [71], [43], [3], [41]. Most of the implementations were hardware-controlled and truly dedicated to FFT only; no other signal processing operations could be performed with these systems.

Hardware control could provide the maximum performance the raw hardware could achieve for a single signal processing application. However, it provided less flexibility. In order to obtain flexibility to cover a wider variety of signal processing applications, the microprogramming techniques developed in the 1970s were applied to the signal processing systems [96], [181], [195]. Microprogramming turned a single-function signal processor into a multi-function signal processor that could handle many signal processing algorithms [4].

A vector processor (or array processor), called AP-120B, appeared in the late 1970s [189], [34]. It was attached to a general-purpose computer to accelerate the performance of vector operations (particularly, addition and multiplication). It included the pipelined adder and multiplier. It was widely used for scientific and engineering applications. The pipeline vector processing techniques were also applied to signal processors [22], [28], [60], since many signal processing algorithms were amenable to vector operations.

With the advances in semiconductor technologies, digital signal processor (DSP) chips appeared in the 1980s [113]. One of the common features in their architectural design is the arithmetic capability for the computations in the form of $A \times B + C$, which are frequently found in the signal processing applications. These are used for relatively small-scale signal processing applications, including telecommunications applications. One of the examples is the mSP32 chip [178]. It is an experimental single-chip 32-bit VLSI signal processor, including a 32-bit floating-point ALU (FALU) and a 32-bit floating-point multiplier (FMULT) to support the floating-point computations in the form of $A \times B + C$.

There have been real-time computing systems entirely dedicated to a single real-time mission, such as an architecture for radar signal processing [65], an architecture for real-time aerospace applications [149], a multiprocessor architecture for industrial process control [92], a distributed architecture for industrial control [166], a fault-tolerant architecture for commercial transport aircraft control [187], and a system architecture for the space shuttle avionics [31].

FFT is still of great importance and a time-consuming computation in the signal processing area. The recent design of FFT processors is more sophisticated than that of the predecessors [15], [162].

Since the dedicated architecture is application-specific, it works very well for the target applications but doesn't work well for the other applications. Our goal is not to design another dedicated architecture for the target class of real-time applications, but to design a more general architecture that can compete with the previous dedicated architectures in performance for the target applications.

3.2 Pipeline Architecture

The basic concepts and techniques of the pipeline architecture are described by C. V. Ramamoorth and H. F. Li in [156] by P. M. Kogge in [93], and by K. Hwang and F. A. Briggs in [85]. Pipelining is one of the major architectural attributes of the modern computing systems. It is a form of parallel processing. It divides a computation process into stages, each of which performs a partial computation, sends the partial results to the next stage, and takes the partial results from the preceding stage. As water flows through a physical pipeline, tasks flow through the pipeline at a rate at which new entries are fed into the input of the pipeline. Tasks in the pipeline are partially processed at each stage. Since different tasks stay at different stages at one time, tasks are processed in parallel. A single task becomes complete when it comes out through the pipeline stages. The execution time of a single task is the total amount of time it resides in the pipeline. However, each task seems to be complete at a rate of new entries fed into the pipeline, which is much higher than the rate at which a non-pipelined processor can perform the task one by one sequentially.

It is an ideal state that tasks continuously flow through the pipeline stages; in this case the pipeline operates at its maximum rate. However, there are hazards that prevent tasks from continually entering the pipeline. Depending on the types of hazards, the processor may delay some tasks at some stages or invalidate all the tasks in the pipeline in order to maintain the consistency of the processor state. They degrade the performance of the pipeline. Many design techniques have been developed to keep the pipeline performance as close to the maximum possible performance as possible in the presence of hazards. There are two general categories of hazards: structural and data-dependent hazards [93, Chapter 1, pp.1–20]. The structural hazards are cases where two different tasks attempt to use the same stage at the same time. The data-dependent hazards occur when a task at one stage determines whether or not data may pass through other stages.

There are two types of pipelining: instruction and arithmetic. Instruction pipelining is for instruction execution. In general the execution of an instruction can be roughly divided into the stages for instruction fetch, instruction decode, operand fetch, and execution. Arithmetic pipelining is for arithmetic computation. For example, a floating-point addition can be divided into the stages for alignment, fraction addition, and normalization, and a floating-point multiplication into the stages for fraction multiplication, exponent addition, and normalization.

We briefly review several pipeline architectures.

In the early 1960s, the IBM System/360 Model 91 was developed as a general-purpose computer with instruction and arithmetic pipelining for floating-point addition, multiplication, and division [9], [8]. Since then, the instruction pipelining techniques for general-purpose computers have been established through the development of the computers in the IBM System/370 family [30], the IBM System/370-XA family [141], and the IBM ESA/370 family [153]. Instruction pipelining is widely used in almost all the commercial computers. See, for example, [45] for the VAX 8600.

Two of the early vector processors are the CDC STAR-100 and the TI ASC. In the late 1960s, the CDC STAR-100 was designed for vector operations and had two arithmetic pipeline processors for floating-point addition and multiplication [78]. In the early 1970s, the TI ASC (Advanced Scientific Computer) was developed for scientific applications and had a high degree of pipelining for both instruction execution and floating-point arithmetic [185].

The scientific attached processors were popular in the late 1970s. The AP-120B is one of them. It was developed as a back-end pipelined arithmetic processor attached to a host computer to accelerate the execution of vector operations [189], [34]. It had a pipelined floating-point adder and multiplier which could operate in parallel. However, it did not have vector instructions; instead, long instructions containing concurrent microoperations were used to specify the parallel operations within the processor. The FPS-164 evolved from the AP-120B [85, Chapter 4, pp.233–324]. It improved the performance over the AP-120B by extending precision (64-bit floating-point operations, instead of 38-bit floating-point operations in the AP-120B) and memory. The IBM 3838 was also designed as a vector processor to be attached to IBM mainframe computers [93, Chapter 4, pp.134–173], [85, Chapter 4, pp.233–324].

The Cray-1 is one of the first modern vector pipelined supercomputers [161], [85, Chapter 4, pp.233–324], [80, Chapter 2, pp.68–143], [121]. It requires a front-end computer like a VAX or an IBM mainframe as a system manager. The Cray-1 contains eight vector registers, each 64-element register (64 bits/element), and 12 functional pipeline units, which are organized into four groups: address, scalar, vector, and floating-point units. All of the functional units can operate in parallel. All integer arithmetic is performed in 24-bit or 64-bit form, and all the floating-point arithmetic is performed in 64-bit form. Two pipelined vector operations can be *chained* together to send the result data stream from a pipeline unit to one vector register simultaneously to another pipeline unit as the operand stream. Pipeline chaining of two vector operations can produce two combined results per machine cycle.

The Cyber-205 is also one of the first modern pipelined supercomputers [85, Chapter 4, pp.233–324], [80, Chapter 2, pp.68–143], [121]. The Cyber-205 contains one scalar arithmetic unit and a vector processor which can have one, two, or four floating-point arithmetic pipelines. The scalar arithmetic unit has five independent functional pipelines for add/subtract, multiply, log, shift, and divide/sqrt operations on 32- or 64-bit scalars. The vector arithmetic pipeline can perform add/subtract, multiply, divide, sqrt, logical, and shift operations on 32- or 64-bit vector operands. Each vector pipeline is directly connected to the main memory without vector registers.

The Japanese supercomputers that followed Cray-1 and Cyber-205 are also pipelined vector processors. They are, for example, Fujitsu FACOM VP-200 [129], [79], Hitachi HITAC S-810 [140] , and NEC SX [94].

When the IBM 3090 system was introduced as a mainframe computer [179], the vector facility was also introduced [27], [142]. The vector facility is optional to the standard IBM 3090 system and can be viewed as an addition to the instruction execution part of the base machine. 171 new instructions were also introduced with the vector facility. The vector facility contains a set of vector registers and two vector pipelines, one multiply/divide pipeline and one arithmetic and logical pipeline. Each of the pipelines can produce one result per machine cycle, except for divide operations. The two pipelines can be chained together to produce two vector operations per machine cycle. For performance evaluation of the vector facility, see [39], [36], [122], and [186].

As discussed above, a variety of pipelined computers have been developed and improved for almost three decades. The pipeline techniques are the established techniques used in almost all recent computers including microprocessors. As indicated by the development of various vector computers, pipelining is very suitable for vector operations. Since an application in our target class contains many vector operations, our architecture should be based on pipelining.

3.3 Parallel Architecture

The parallel architecture performs multiple operations, synchronously or asynchronously, at the same time. It has many variations. In this section, we discuss the MIMD (Multiple Instruction streams, Multiple Data streams) architecture, which was defined by M. J. Flynn in [58]. There are several books and papers on MIMD parallel architectures. See, for example, [85], [80], [14], [51], [163], [86], [68]. We outline the features and problems of the MIMD architecture.

In the 1970s, many of the commercial mainframe computers added a multiprocessing capability as in the IBM System/370 Model 168, the CDC Cyber 170, the Honeywell Series 60 Level 66, the UNIVAC 1100 Model 80, and the Burroughs B7700 [163]. In these multiprocessor systems, the main memory was shared among the processors. The number of processors was limited to 2 to 4. These multiprocessor systems were designed to execute different tasks on different processors in order to achieve higher throughput, rather than to speed up the execution of a single task. They can be viewed as a parallel extension to a multiprogrammed uniprocessor system in that tasks that had been executed in a time-sharing manner in a uniprocessor system were executed in parallel on different processors. Although the internals of the operating systems were changed to accommodate the multiprocessing capability, no changes were made to application programs that were executed as tasks. The major changes to the operating systems include the mechanisms to synchronize processors competing for system resources like disks. The synchronization mechanisms used some hardware-supported means for interprocessor communications. In the IBM System/370 Model 168, for example, the SIGP (Signal Processor) instruction and the TS (Test and Set) or CS (Compare and Swap) instruction were used for that purpose. The processor executing the SIGP instruction interrupts the other processor to get its attention for communications. The TS or CS instruction performs an atomic memory access and is used to reserve a memory location associated with some system resource, which is protected against unauthorized accesses.

Since UNIX operating systems have become popular, many multiprocessor UNIX systems have been developed [68]. They have more processors than the multiprocessor mainframe systems developed in the 1970s. The Sequent Balance (8000 and 21000) system contains 2 to 30 microprocessors (NS 32000 series) and runs the DYNIX operating system supporting the 4.2 BSD and System V Unix environments [175]. Each processor has a private write-through cache memory. The bus-watching logic at each cache continuously monitors write cycles on the system bus, comparing the write addresses to the internal cache state. When the logic detects a hit, the cache controller invalidates the affected entry.

The Encore Multimax is another example of a shared-memory Unix multiprocessor system [192]. It can be configured with 2 to 20 processors and up to 30 I/O channels. Each processor has a private cache memory with a write-back protocol, which defers the store of updated portions into memory until the last possible moment. In order to maintain the cache consistency, cache entries are invalidated when there are updates in one cache. The invalidation of the cached copies is done by sending a special read request which is issued the first time a write or read-modify-write is attempted to a location.

The BBN Butterfly Processor is also a UNIX multiprocessor system [165], [82]. Unlike the previous two UNIX multiprocessor architectures, however, it has a memory distributed over processing nodes. It can have 2 to 256 processing nodes, which are connected by the butterfly switching network, an indirect binary n-cube packet-switching network. Each processing node contains a CPU and a local memory. Memory accesses by a CPU are either local or remote. Local memory accesses are made to a local memory; remote memory accesses are made through the butterfly switching network to a memory on a different processing node. Experience with the Butterfly processor is reported in [126], [136].

There have been many experimental architectures developed for some research purposes. The early experimental parallel architectures are C.mmp [197], [196] and Cm* [67], [66], both developed at the Carnegie Mellon University. The C.mmp was developed in the early 1970s to examine the feasibility of a large multiprocessor computer for artificial intelligence research, and, from the viewpoint of computer architecture, to study the parallel architecture with the hierarchically structured processor clusters. Up to 16 of processors could be connected to up to 16 shared memory modules through a 16×16 crosspoint switch. The Cm* consists of 50 processor-memory pairs called Cm's, which are connected by a hierarchical, distributed switching structure. It is divided into five clusters of up to 14 Cm's each. The clusters are connected via intercluster buses. Experience with C.mmp and Cm* is reported in [89].

The S-1 system is also an experimental multiprocessor system aimed at the processing performance of over 10 times the computational power of the Cray-1. It consists of 16 processors that are connected to 16 memory banks of the main memory through the 16×16 crossbar switch. The Burroughs Scientific Processor or BSP is another example of an experimental multiprocessor system [103]. It consists of 16 processors that are connected to 17 memory banks via two data alignment networks. The system has a five-stage memory-to-memory data pipeline, plus earlier instruction-setup pipeline stages, one for input and the other for output. The memory-to-memory data pipeline is formed by five stages for fetch, align, process, alien, and store. It is unique in that it uses 17 memory banks to achieve conflict-free memory accesses.

The Denelcor Heterogeneous Element Processor or HEP is yet another example of a shared-memory MIMD architecture [95]. It consists of one or more pipelined MIMD Process Execution Modules (PEMs) which share a common memory; each PEM is an MIMD computer. There is no qualitative difference in the way processes are created and managed or in the way in which they communicate in a single-PEM system versus a multi-PEM system. The PEMs and the shared memory banks are connected by a pipelined message-switched interconnection network. One of its unique features is the full/empty synchronization mechanism. Other experimental multiprocessor systems include the NYU Ultracomputer [70], the UIUC Cedar [102], and the IBM RP3 [151].

The pipelined supercomputers have been adding a multiprocessing capability. The Cray X-MP/1 has two CPUs, each of which is very similar in structure to the Cray-1 CPU [85, Chapter 9, pp.643–731]. It was designed for multitasking applications; it can run independent tasks on two processors. Discussions on the multiprocessor supercomputers are presented by K. Hwang in [84].

Multiprocessor systems have been developed to achieve the high performance that a uniprocessor system can not achieve. The n-processor system is expected to achieve the performance n times the performance a uniprocessor system can achieve. This is true if n processors execute independent tasks without sharing anything. However, tasks usually share some system resources, such as code and data. These tasks must be synchronized to maintain the consistency of the system state. Synchronization has been one of the focal issues in the multiprocessor system design [64], [148], [47]. Many techniques for efficient synchronization have been studied and implemented [46]. In general, the synchronization overhead in the MIMD architecture is not small, because the constituent processors operate asynchronously and each processor doesn't have any information about the states of the other processors without performing explicit interprocessor communications. Even in interprocessor communications, there is no guarantee that the processor can quickly respond to a request by the other processor. Due to the large synchronization overhead, the grain size of parallel processing has to be large enough to pay off the overhead; that is, the execution between one synchronization point and the other has to be as long as possible. To exploit fine-grain parallelism, commonly found in solving a single problem with multiple tasks, the MIMD architecture has to provide a synchronization mechanism with a small overhead. It is one of the recent research topics.

The other issues in the multiprocessor architecture include memory accesses. The memory that can be physically or logically shared by processors is accessed through a shared buse or a switching network. It is known that there will be a hot spot when processors try to access a particular location of memory, and it significantly degrades the system performance [152]. Cache memory is another important issue in memory accesses. When processors frequently update a particular area of memory, many cache invalidation operations occur and degrade the system performance. It is a research topic to find an efficient invalidation scheme that keeps coherency. See [47] for more detail.

The MIMD architecture is a fairly general architecture. It is too general for our target class of real-time applications. We can simplify the design of the system architecture and avoid some of the problems in the design of the MIMD architecture by exploiting the characteristics of the target applications. Since the target applications are based on vector operations, which can be performed synchronously, asynchrony which comes along with the generality of the architecture is undesirable. Asynchrony in system operations requires explicit interprocessor communications to get the information of the other processors, which degrade the system performance. Although much fine-grain parallelism can be found in vector operations, the MIMD architecture can not exploit it due to the high cost of synchronization.

3.4 Massively Parallel Architecture

The massively parallel architecture is classified as an SIMD (Single Instruction stream, Multiple Data streams) architecture according to Flynn's taxonomy [58]. It consists of many simple processing elements, all of which perform an identical operation under the control of a single instruction. It is also called the processor array architecture because of its organization. In many cases a single instruction implies a single operation for all the processing elements. In general it shows a good performance when it performs a single operation on a large amount of data at the same time.

The Illiac IV system was the first SIMD supercomputer developed at the University of Illinois in the 1960s [25], [81]. It consists of 64 processing elements which are interconnected as a two-dimensional mesh network (an 8 array). Each processing element contains registers, arithmetic units, and data links to its four neighbors. Data are represented in either 64- or 32-bit floating-point, 64-bit logical, 48- or 24-bit fixed-point, or 8-bit character form. Each processing element can hold vectors of operands with 64, 128, or 512 elements. All the processing elements are under the control of a control unit that transmits 32-bit instructions to them. The system has been used for numerical weather forecasting and nuclear engineering research.

The Massively Parallel Processor or MPP was developed at Goodyear Aerospace in the early 1980s [154], [17]. It consists of 16,384 processing elements, which are arranged in a 128×128 array. Each processing element is a bit-serial processor that performs only single-bit operations, but can handle data of any length by processing them bit by bit. It can communicate with neighboring processing elements as in the Illiac IV. It spends time in proportion to the number of bits in data when it processes multiple-bit data. The MPP contains staging memories that serve as data buffers for the array of processing elements. They also have a capability of reordering data, which is required if the order of data input is not the same as the order of processing. The DEC VAX-11/780 was attached to the MPP as a host computer when it was delivered to NASA.

The Thinking Machines' Connection Machine is also a massively parallel architecture [77], [180]. Unlike the Illiac IV and the MPP, the processing elements in the Connection Machine are interconnected in the hypercube form (see the following section for the hypercube architecture) for interprocessor communication. The first implementation of the Connection Machine is CM-1 with 16K to 64K data processors. When the CM-1 is configured to have 64K data processors, four blocks of 16K data processors with a sequencer are connected by a 4×4 cross-point switch called nexus. The data-parallel I/O system connects processors to peripheral mass storage (called the Data Vault) and graphic display devices. The applications for the Connection Machine include computer vision, VLSI design and circuit simulation, molecular dynamics, and so forth.

The similar massively parallel architectures include the ICL DAP [57], the CLIP [62], the STARAN [16], and the RPA [160].

The massively parallel architecture is suitable for the applications that perform one relatively simple operation on a large amount of data. Image processing is a good example application for it, since Many systems with this architecture connect each processing element with a few of its neighboring processing elements. This architecture can provide a large amount of fine-grain parallelism but limited interprocessor communications. Our target real-time applications could be handled by this architecture, but it is not straightforward. We can exploit the characteristics of the target applications, especially those of multiple-bit vector operations.

3.5 Systolic Architecture

The concept of systolic arrays was first presented by H. T. Kung and C. E. Leiserson in [104]. It was originally proposed for VLSI implementation of some matrix computations. Since then, several papers on systolic arrays were published [105], [115], [61], [106], [108]. S.-Y. Kung and his group proposed the similar concept called the wavefront arrays [107], [108], [109].

The systolic architecture is based on a systolic computing model that derives computational efficiency from parallel and pipeline processing. The systolic array architecture consists of many (relatively small) processors arranged in various shapes by connecting them with communication links. Data enter the systolic array through input elements and are propagated to neighboring processors for further processing. Data move along a fixed direction in which a link exists between neighboring processors and in a periodic manner. As data flow through a sequence of processors, operations are applied on them in each processor. Each processor in the systolic array repeats the same operations on different data when they pass by it. Operations are all regular and synchronous. A stream of data is processed in a pipeline fashion.

The systolic architecture provides not only a parallel architecture but also a design methodology for implementing VLSI algorithms. The VLSI algorithms that can be implemented with systolic arrays are characterized by the systolicity in their computations. There are many well-know algorithms that can be implemented with systolic arrays: matrix computations, polynomial computations, graph algorithms, signal and image processing algorithms including FFT and convolution, and so forth. When we apply the systolic array to a problem, we need to systolize the algorithm for solving the problem. It is a mapping problem: how to map an algorithm on an architecture. S.-Y. Kung presented an approach to this mapping problem in [108]. His approach gives a method to convert computing networks representing computations into systolic arrays or wavefront arrays. H. F. Li and R. Jayakumar rigorously characterized the systolic structures and presented an approach to deriving the systolic algorithms in [118].

The systolic architecture is very useful for our target class of real-time applications, although it cannot be directly applied to the architecture to support the applications, because each application may contain a variety of vector operations, and all the operations are not necessarily predefined at the time of architectural design. The useful features of the systolic architecture are parallelism, pipelining, and regularity.

3.6 VLIW Architecture

The concept of the VLIW (Very Long Instruction Word) architecture was presented by J. A. Fisher in [54], [55]. The VLIW architecture is characterized by its very long instructions. It consists of multiple processors and one program counter which indicates the address of a machine instruction to execute. In this architecture there are parallel operations but only a single thread of control. Each instruction contains many operation fields to control each of the individual processors. A single instruction determines the operations of all the processors; the operation for one processor may be different from that for another processor, depending on the corresponding operation fields of the instruction. The instruction also controls all the communications between the processors. The very long instruction word is very similar to the long horizontal microcode word, which contains may operation fields to control many hardware components in the processor at the same time.

The effectiveness of the VLIW architecture heavily depends on the VLIW compiler that compiles a program into a sequence of very long instruction words with efforts to pack as many operations as possible in a single instruction. The first compiler developed for the VLIW computer is Bulldog [50]. It uses a global compaction technique called trace scheduling, originally developed for microcode compaction [53], in order to generate very long instruction words from a sequential source code. The trace scheduling technique heuristically guesses the run-time control flow of a program and chooses the most likely forward execution stream from the possible streams. Guesses are made for conditional branches. The compiler also generates the code to move from the main execution stream to the other stream when the guess turns out to be incorrect at run time. The compiled code runs with the maximum performance if the guesses are correct; it degrades the performance of the machine if the guesses are incorrect, but still produces correct results.

Parallelism available for the VLIW architecture is reported in [131]. Measurements were made on 22 Fortran programs, most of which were numerical analysis programs. They showed that the available global speedups ranges from 4 to 988 with the VLIW architecture. The programs that exhibited the largest parallelism operate on arrays. Array processing presented great opportunities for parallelism. See [131] for more detail.

The VLIW architecture may be classified as an SIMD (Single Instruction stream, Multiple Data streams) according to Flynn's taxonomy [58], but it is clearly different from the massively parallel architecture or the processor array architecture, in which the same fields contained in a single instruction are used to control all the constituent processors. Although they assumed that a processor controlled by an operation field of a very long instruction is something like a RISC (Reduced Instruction Set Computer) processor, we need not limit the processor to it; we can use more primitive arithmetic units instead of RISC-type processors. Moreover, as indicated by the parallelism measurements, the VLIW architecture is suitable for vector operations. Therefore, the basic concepts of the VLIW architecture, instead of the architecture itself, is useful for our target class of real-time applications.

3.7 Hypercube Architecture

The hypercube architecture is based on the hypercube connection of processing elements, each including a processor and a memory, and coordinates their computations by sending messages to each other [167], [12], [74]. It differs from a shared-memory multiprocessor, in which processors are connected to a shared memory through a switching network or a common memory bus. Each processing element of the hypercube architecture operates as an independent computer. The main feature of the hypercube architecture is in its way of interconnecting processor elements.

The hypercube topology has several interesting properties. One of the most interesting property is the communication path length. Any message sent by a processor element can reach the destination in no more than $\log n$ hops in the n-dimensional hypercube architecture. That is, the maximum length of the path (the number of edges on the path) a message may follow is $\log n$. It is a good property for implementing a variety of algorithms that require many communications between processing elements. When many processing elements send messages, messages may collide at one node for a single edge. Then all but one messages are delayed for the use of the edge, or they may be sent along the other edges and take more hops than required theoretically.

Each processing element is an independent computer that runs its own copy of an operating system and operates asynchronously. When programs run on processing elements to solve a single problem, they must synchronize at some points. If there are many such synchronization points, then the performance of parallel execution is destroyed. Since the synchronization overhead is not small in this type of architecture, it is suited for coarse-grain parallel execution, not for fine-grain parallel execution. In order to obtain asynchronous coarse-grain parallelism, we have to make a program so that the number of synchronizations between processing elements should be kept minimum, while the number of operations between synchronization points should be kept maximum.

A message goes through processing elements on its path to the destination. Each processing element has to use some CPU time to route messages. The more messages come, the more time the processing element has to spend for routing, degrading the performance. Special hardware can be added to a processing element not to interrupt tasks running on it. However, when the tasks come to a synchronization point, they have to send or receive messages. Still, its performance is affected by the messages passing by that processing element. There is no way to avoid the degradation of performance due to routing messages in this architecture.

From the implementation point of view, the implementation of the n-dimensional hypercube is limited by the number of links each processing element has, if n is large. For example, if $n = 10$, then each of $2^{10} = 1024$ processing elements has 10 links. If each link is 8 bits, 16 bits, 24 bits, or 32 bits wide, then the total number of bits for each processing element is 80, 160, 240, or 320, respectively. Since each connection is one-to-one, the total number of wires (one wire per bit) amounts to 40,960 for 8-bit links, 81,920 for 16-bit links, and 122, 880 for 24-bit links, and 163,840 for 32-bit links. Since some control signals and parity bits are usually associated with each link, the total amount of wiring is more than that number for each case. If we use a single-bit link, the amount of wiring is small, but it takes more time to transfer data.

In general, the hypercube architecture is expected to show an efficient communication capability because of the property of the hypercube connection. However, it is not an easy task to control the precise timings, because of the asynchronous parallel execution and influence of message routing. Moreover, it is the coarse-grain architecture, which is not suited for arithmetic-level parallelism. It is very difficult to implement the architecture with a multiple-byte link for multiple-byte communications. Therefore, it is not a good architecture for our target class of real-time applications.

3.8 Data-Flow Architecture

The data-flow architecture is based on a data-driven computing model, where a computation is initiated as soon as the required operands become available [44], [177], [11], [117]. It is quite different from the traditional von Neumann architecture in that its computation is driven by the availability of data, whereas the computation in the conventional von Neumann architecture is controlled by a program consisting of instructions. A von Neumann architecture executes instructions synchronously using a centralized control, while a data-flow architecture has no centralized control over computations and executes instructions asynchronously when their operands are ready. Since there is no predefined sequence of instructions, a high degree of potential parallelism can be exploited in the data-flow architecture. Several experimental data-flow machines have been constructed and evaluated (see [177] and [11]).

A program, called a *data-flow program*, for the data-flow architecture, is represented by a directed graph where the nodes denote operations (addition, multiplication, etc.) and the edges denote data dependencies between operations. One of the programming languages for the data-flow architecture is VAL [124]. It is a high-level, function-based language designed for the data-flow architecture. VAL provides implicit concurrency by using functional language features, which prohibit all side effects. It is based on a single-assignment rule which allows a variable to take only one value in a single expression and not to depend on itself.

The data-flow architecture is claimed to eliminate the von Neumann bottleneck that limits potential parallelism in computations. However, it creates new problems. One of them is the overhead per token, including the communication overhead for packing tokens, routing tokens, unpacking tokens, etc. If most of the operations are small, the total overhead of tokens dominates the execution of a program in time and space. The implicit parallelism can not be exploited for high performance.

It is pointed out that it is very difficult to design a data-flow architecture that can handle a large array of data, which could be easily stored in the main memory in the conventional architecture. Large arrays of data are required to store large vectors and matrices that are commonly used in scientific and engineering applications. The data-flow architecture doesn't have a main memory; it doesn't have a place to store a large volume of data. Data can be distributed in pieces over processing elements. But it raises a heavy communication problem.

The other problem is that the data-flow architecture is based on run-time scheduling of operations, whereas the conventional von Neumann architecture is based on compile-time scheduling of operations. For the conventional architecture, the compiler can schedule instructions at compile time; there is no instruction-level scheduling at run-time. However, the data-flow architecture schedules instructions at run-time, which could be scheduled by the compiler at compile-time. We haven't seen any non-trivial problems that require run-time scheduling of instructions. Therefore, the data-flow architecture is a costly solution with respect to scheduling.

The data-flow architecture doesn't seem to be a fully developed and established architecture. It may have overcome the von Neumann bottleneck, but it has created many other technical problems that are not problems in the traditional von Neumann architecture. It has not essentially broken the von Neumann "barrier." The data-flow architecture itself is premature to apply to our target class of real-time applications that require extensive vector operations.

3.9 RISC Architecture

The *RISC* or *Reduced Instruction Set Computer* architecture is the architecture that improves performance by simplifying machine instructions and executing as many instructions as possible within a CPU without referencing main memory [145], [155], [75], [147], [183].

The traditional computer architecture had been moving towards the integration of more functionality in machine instructions with the help of microprogramming. This type of architecture is called the *Complex Instruction Set Computer* or *CISC* architecture. As discussed in [146], the design of the CISC architecture is not always cost-effective with the advances in VLSI technologies, since the design time is expanded and design errors are increased due to the high design complexity. Although the complex instructions can provide more functionality than the simple instructions, it is not an easy task for compilers to generate the complex instructions. If compilers cannot generate these complex instructions, the part of the architecture dedicated to the functions for the complex instructions is not used. The chip area used for the functions for the complex instructions can be more effectively used for more useful functions like a larger set of registers and on-chip cache memory in the RISC architecture. Thus, the basic idea behind the RISC architecture is the simplicity in design and the high performance with more commonly used system-wide functions.

The RISC architecture requires a good optimizing compiler. The compiler for the architecture has to perform extensive optimizations to reduce the total size of code; otherwise, the size of code tends to be much larger than that for the CISC architecture, since the RISC architecture provides a set of simple instructions. The optimizing compiler also has to keep as many operations as possible within the processor without referencing the external memory, using a larger set of registers provided by the RISC architecture. Register allocation is very important for this purpose. The optimizing compiler also has to deal with the pipeline break with branch instructions, which may change the instruction stream and require the pipeline flush, leading to degradation in performance.

The RISC architecture is now getting popular in the design of high-performance commercial microprocessors: Motorola 88000 [6], MIPS R3000 [157], and Hewlett-Packard PA [114], to name but a few. The RISC architecture is basically a general-purpose processor architecture to be implemented on a VLSI chip. It provides no special advantages for vector operations required for the target class of real-time applications. However, it can be used for general system control.

3.10 Superscalar Architecture

The superscalar architecture is the architecture that improves performance by concurrent execution of scalar instructions [88]. The superscalar techniques can apply to both RISC and CISC architectures, since they mainly concern the processor organization, independent of the instruction set and other architectural features. One of the attractive features of the superscalar architecture is the code compatibility with the existing architectures. In principle, it does not require any modification of the code written for a RISC/CISC processor, if the superscalar techniques are applied to the processor organization.

The basic idea behind the superscalar architecture is to exploit the instruction-level parallelism with the lookahead capability to examine instructions beyond the current point of execution. The superscalar processor fetches a subsequence of instructions from the complete sequence of instructions stored in memory and checks to see if there are instructions in the subsequence which can execute in parallel without any conflicts. If the processor finds those instructions, it issues them to multiple functional units. Otherwise, it serializes the instruction execution to avoid the inconsistency in the processor state.

A sequence of instructions can have three factors that limit the performance of a superscalar processor: data dependencies, procedural dependencies, and resource conflicts [88]. A data dependency exists if an instruction uses a value produced by a previous instruction. In this case, the instruction has to wait until the previous instruction produces its result value. A procedural dependency exists if a conditional branch instruction occurs in a sequence of instructions. In this case, the processor can not execute the instructions following the branch instruction until the branch instruction is executed. A resource conflict exists if an instruction uses the same resource that the previous instruction uses. In this case, the processor has to delay the instruction until the previous instruction finishes and releases the resource. All of these factors cause the serialization of instruction execution, limiting the performance of the processor. See the studies on the available instruction-level parallelism by N. P. Jouppi and D. W. Wall in [90] and by D. W. Wall in [182].

The superscalar architecture is very useful when we design a new processor that executes the code written for a scalar processor with higher performance. It is also suited for executing the instructions generated by the traditional compilers in a traditional execution environment. However, it is not suitable for our target class of applications, because the our goal is to design the real-time system architecture to meet timing requirements imposed by the real-time applications, not to speed up the execution of existing programs. We have no software compatibility problems, for which the superscalar architecture works well. The superscalar architecture is a solution to speeding up the ordinary programs involving many irregular operations, whereas our target applications are based on vector operations which are very regular. We can exploit the regularity of vector operations for architectural design.

Chapter 4

DRA Design Concepts

This chapter presents the design concepts of the real-time computing system architecture for the target class of real-time applications. It also presents theoretical models for performance and execution.

4.1 Design Philosophy

We are interested in the design of a real-time computing system architecture that is suitable for the target class of real-time applications described in Chapter 2, that is, the class of real-time applications that handle a large volume of periodic real-time data, on which a variety of vector operations are performed.

In Chapter 3, we have reviewed the major existing computing system architectures. Some of them are not quite suitable for the target class of real-time applications. They include the dedicated, parallel, massively parallel, hypercube, data-flow, and superscalar architectures. The others (pipeline, systolic, VLIW, and RISC architectures) have some features suitable for the target applications. Especially, the pipeline architecture matches vector operations very well, and is the basic architecture for the target class of real-time applications. However, the pipeline structure employed in the commercial supercomputers is too general; we can optimize it with the characteristics of the target applications. Synchronous pipeline computations in the systolic architecture are also a useful feature for the target applications. The feature of the VLIW architecture can be used in our architecture without worrying much about switching of instruction streams, because vector operations are provided in the loop constructs in many cases. The RISC architecture is useful for the control of vector computations.

The major timing constraints come from the periodicity of the input data. All the required tasks for a set of input data must finish until the next set of data comes in. For such a timing requirement, the statistical or probabilistic behavior of the system is undesirable. If there are many statistical factors in the architecture, it is very difficult to ensure the timing correctness of the system, and it is also difficult to trace the timing errors when they ever occur. Many of the state-of-the-art techniques in computer architecture can statistically provide high performance. Cache memory is a typical example of the statistical system components. It can significantly speed up memory references with more than 90% hit ratios. However, its behavior heavily depends on the memory access patterns of application programs and the operating system. It also depends on the history of memory accesses.

Since the existing computing systems are not suitable for the target class of real-time applications, we need to design a new architecture that is capable of handling the target real-time data. First, the architecture has a system structure that flows a large volume of real-time data smoothly without any blockage. Since the volume of data is large, it can process data in an on-line fashion; it has to store data in a memory. In order to flow data freely, the system needs to have some form of pipeline processing. All the pipeline stages must be built around memory. The memory need to be organized to have four banks that can operate independently. The first of them is used to store a set of input real-time data, the second to store a set of data for processing, the third to store a set of result data after processing, and the fourth to store the data for output. They need to operate at the rate of real-time data input, so that real-time data flow through the system like a free-flowing water. In order for the four memory banks to operate independently, they have to be connected to four data buses so that each memory bank can use one of the data buses without any conflict.

At one of the system-level pipeline stages, vector operations are performed on the input data in an execution unit. It is well known that vector operations have an affinity for pipeline processing. It is not reasonable to have a pipeline for each computation aspect of the application, since it requires an unreasonable amount of hardware. Computational resources must be shared among all the computational aspects of the application to make the amount of hardware reasonable. No matter how complicated a vector operation is, they can be decomposed into arithmetic-operation-level vector operations, such as arithmetic-logic, shift, and multiply operations. Therefore, in the new architecture, the arithmetic-operation-level functional modules are the computational resources to be shared and to be interconnected to form one or more pipelines for the required computations.

The interconnections of the computational resources need to be changed dynamically for different vector operations. A cross-bar switch could provide all the possible interconnections of the computational resources, but would be very expensive. A less expensive switching network is desirable for changing the interconnections. The detailed analysis of the vector computations required by the application can determine the set of computational resources (number and kind) and their possible interconnections. The types and numbers of the computational resources could be systematically determined, but it is beyond the scope of this thesis. All the interconnections of computational resources should form pipelines to support for the pipelined vector operations; each computational resource also needs to be pipelined so that the interconnected resources can be pipelined.

Pipeline processing does not involve any statistical or probabilistic behavior; operations are applied on data at each pipeline stage, regularly at each clock cycle. It is easy to check the processing timings in the computation network, because it is predictable when a particular data item comes in and out. Given a set of computational resources, the design of a pipeline for a vector operation determines the processing time, which can be computed from the total number of pipeline stages from input to output. Thus the dynamically reconfigurable computation network addresses the issues of timings as well as those of resource sharing.

The following sections present performance and execution models for the new architecture. The performance models give the theoretical performance achieved by vector operations and pipeline processing, upon which the new architecture is based. They are derived from the well-known results in the computer science and engineering fields. They show the limits of the performance gain with vector and pipeline operations. The execution model gives a theoretical model for execution and presents the programmability of the new architecture based on the dynamically reconfigured arithmetic pipelines. It is derived from the work done by some other researchers. It is important particularly from the software point of view. It includes the techniques to convert a loop construct to a pipeline network. It is also a basis for the possible design tools, which are beyond the scope of this thesis.

4.2 Performance Models

Parallel/Vector Performance Model

Amdahl's Law is well-known as to parallel performance [7]. Define N, s and p as follows: N = the number of processors, s = the time spent by a serial processor on a sequential portion of a program, and p = the time spent by a serial processor on a portion of a program that can be executed in parallel. Then Amdhal's Law says that the speedup S obtained by executing the program with N processors is given by

$$S = \frac{1}{s + p/N} \,, \tag{4.1}$$

where $s + p = 1$. If we substitute p with $1 - s$, then we obtain

$$S = \frac{1}{s + (1 - s)/N} \,. \tag{4.2}$$

This is illustrated in Figure 4.1 with $N = 1000$. It shows that the speedup obtained with 1000 processors is only 500.3, 333.6, 250.2, and 200.2 if the sequential portion of the program

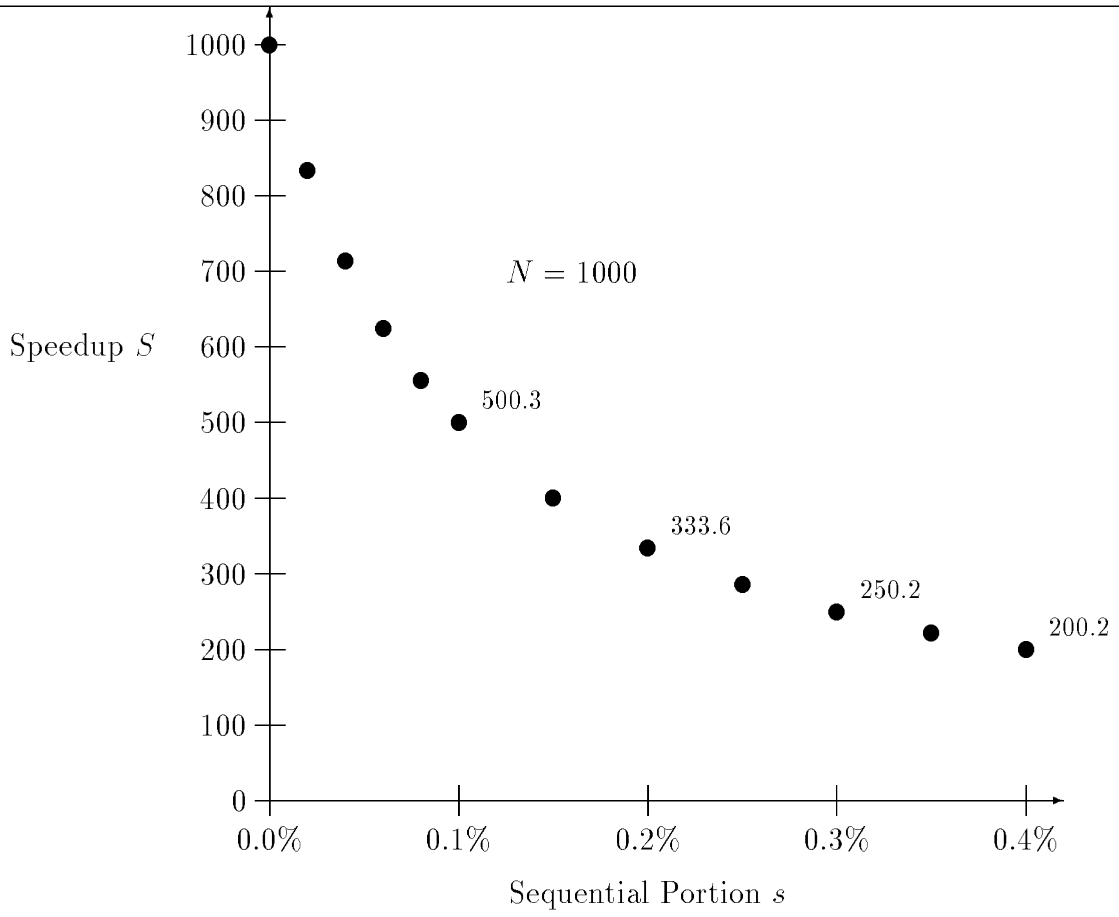


Figure 4.1: Speedup under Amdahl's Law.

is 0.1%, 0.2%, 0.3%, and 0.4%, respectively. Based on this equation, W. H. Ware showed that even a small amount of serial processing can significantly reduce the effectiveness of a multiprocessor [184]. G. M. Amdahl argued in [7] that since parallel performance is limited by the amount of the sequential portion of a program, the single processor approach is the way to go. Although the current trend in computing technologies shows the clear direction towards parallel computing with multiple processors [150], [63], [69], it is an unquestionable fact that the sequential portion of a program limits the performance any parallel computing system can achieve. Many efforts are being made to reduce as much sequential portion of a program as possible.

J. Worlton presented more elaborate discussions about parallel processing in [193]. He used the following model:

$$B = \frac{1}{F_H \cdot T_H + F_L \cdot T_L} , \tag{4.3}$$

where

B	= results generated per unit time,
F_H	= the fraction of results generated in high-speed mode,
T_H	= the time to generate a single result in high-speed mode,
F_L	= the fraction of results generated in low-speed mode, and
T_L	= the time to generate a single result in low-speed mode.

It models a computer that operates in the high-speed and low-speed modes, implying the parallel and sequential modes, respectively. If we consider the effect of the enormously fast high-speed mode, that is, $T_H \approx 0$, then

$$B \approx \frac{1}{F_L \cdot T_L} . \tag{4.4}$$

This indicates that the speed of a computer having two modes of operation is limited by its low-speed mode given the fraction of results generated in that mode.

If we divide by T_H the numerator and denominator of Equation 4.3, then we get

$$B = \frac{B_H}{F_H + F_L \cdot (T_L/T_H)} , \tag{4.5}$$

where $B_H = 1/T_H$. This form of the model shows the effect of the ratio T_L/T_H on B. As the ratio increases, the effect of F_L also increases.

Based on these performance models given by Amdahl and Worlton, we now formulate our performance model.

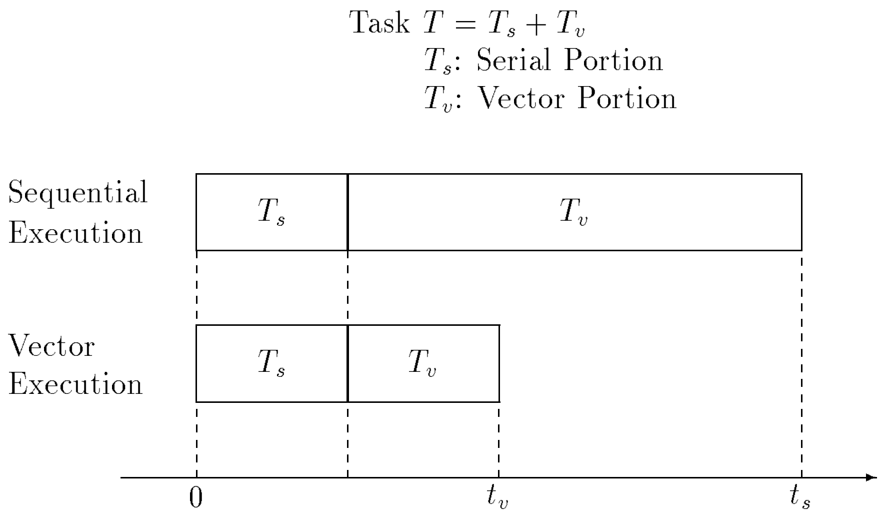


Figure 4.2: Sequential vs. Vector Execution.

Assume that a task T consists of two portions, T_s and T_v, where T_s is the sequential portion of the task that can be performed only by sequential operations, and T_v the vector portion that can be performed by vector operations. Let r_s and r_v be the fractions of the task corresponding to T_s and T_v, respectively, where

$$r_s + r_v = 1 \ . \tag{4.6}$$

Let t_s and t_v be the sequential execution time of T and the execution time of T when T_v is performed by vector operations, respectively. This is illustrated in Figure 4.2.

Let A be the acceleration factor defined as the ratio of vector operations to sequential operations for T_v. Then, the total speedup S obtained by vector operations for T is represented by

$$\begin{aligned} S &= \frac{t_s}{t_v} && (4.7) \\ &= \frac{1}{r_s + r_v/A} \ . && (4.8) \end{aligned}$$

If the acceleration factor A is so large that $1/A$ is negligibly small, that is, $1/A \approx 0$, then we get

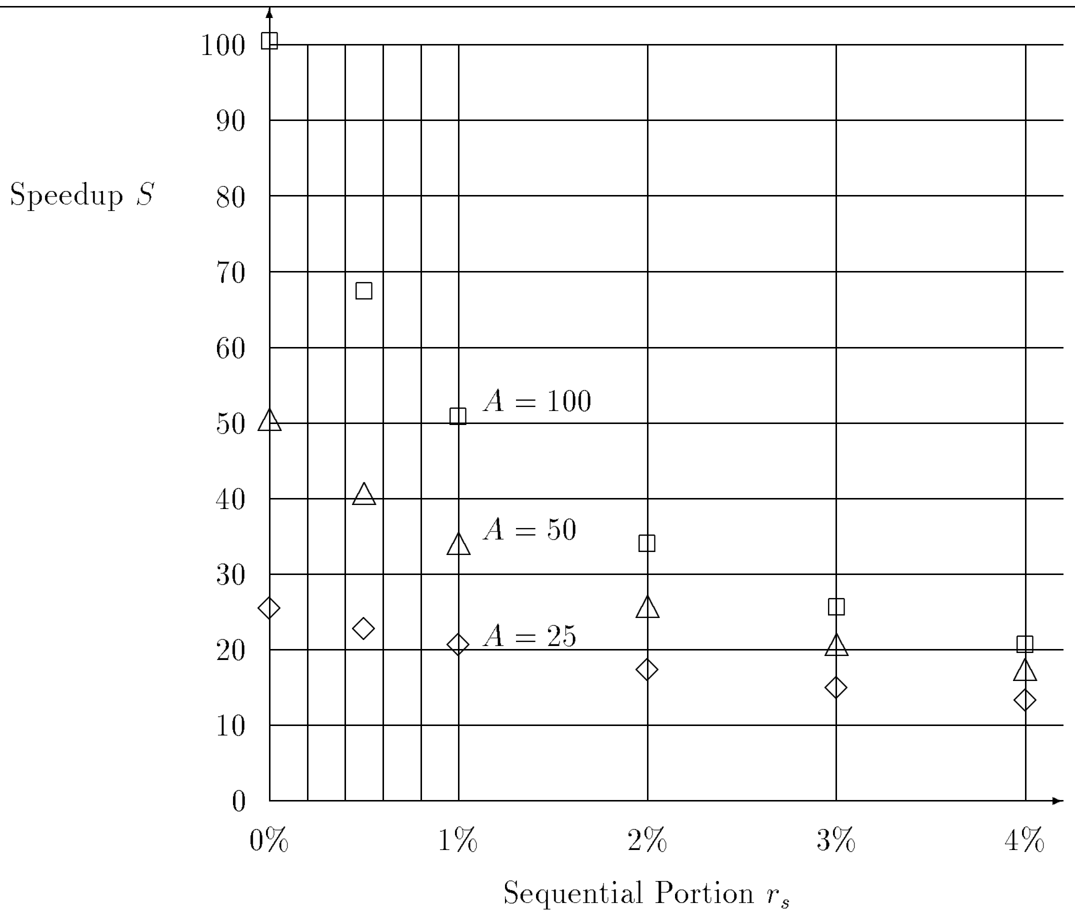


Figure 4.3: Speedup with Vector Operations.

$$S \approx \frac{1}{r_s} \ . \tag{4.9}$$

Figure 4.3 shows the speedup with various values of r_s and A. For example, if $r_s = 0.01$, $r_v = 0.99$ and $A = 100$, then $S = 1/0.0199 = 50.3$. This means that if vector operations, which are 100 times faster than the corresponding sequential operations, are applied to the vector portion of the task, the total execution time of the task is reduced to 1.99% of its sequential execution time, giving the speedup of 50.3. Figure 4.3 also shows that the speedup obtained with a large value of the acceleration factor is not always greater than that with a small value of the acceleration factor, depending on the ratio of the sequential portion. For example, the speedup obtained with $A = 100$ and $r_s = 0.04$ is less than that with $A = 25$ and $r_s = 0.005$ (20.2 vs. 22.3).

When we design a real-time system, we can determine the acceleration factor A to some extent, since it is derived from the system's architectural features and technology. In general, we can increase it by choosing expensive technology and adding expensive architectural features. Given $S = t_s/t_v$ and $r_s = 1 - r_v$, A is represented by

$$A = \frac{1 - r_s}{1/S - r_s} \ . \tag{4.10}$$

For example, if $S = 10$ and $r_s = 0.05$, then $A = 19$. It means that if we need the speedup of 10 for the application with the sequential portion of 5%, we have to design a system that executes the vector operations 19 times faster than the corresponding sequential operations. Since the feasible value of A must be positive, the following condition must be satisfied

$$r_s \cdot S < 1 \ . \tag{4.11}$$

That is, the product of the expected speedup and the sequential portion must be less than 1.

Given the acceleration factor, we may have many options to achieve it in designing the system. Then we choose one of them by taking into account the design factors to optimize the system design.

A very similar argument holds for parallel processing. In this case, we consider a task T consisting of T_s (a sequential portion) and T_p (a parallel portion). We use r_p (the fraction of the parallel portion) instead of r_v, and t_p (the parallel execution time of T_v) instead of t_v. A should be interpreted as the acceleration factor defined as the ratio of parallel operations to sequential operations for T_p. This is illustrated in Figure 4.4, which is very similar to Figure 4.2.

A task T can be processed by parallel and vector processing. Consider a task T that consists of a sequential portion T_s and a vector portion T_v, which in turn consists of a sequential portion T_{vs} and a parallel portion T_{vp}. Define r_s, r_v, r_{vs} and r_{vp} as follows:

r_s	= the sequential fraction of T,
r_v	= the vector fraction of T,
r_{vs}	= the sequential fraction of T_v, and
r_{vp}	= the parallel fraction of T_v,

where

$$r_s + r_v = 1 \ , \tag{4.12}$$

$$r_{vs} + r_{vp} = 1 \ . \tag{4.13}$$

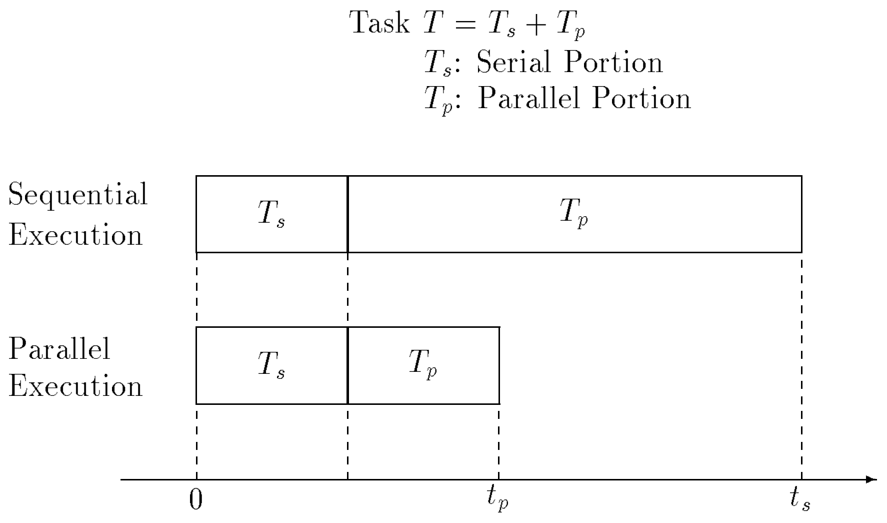


Figure 4.4: Sequential vs. Parallel Execution.

Let t_s, t_v, and t_{vp} be the sequential execution time of T, the execution time of T when T_v is processed by vector processing, and the execution time of T when T_v is processed by vector processing and T_{vp} is processed by parallel processing. This is illustrated in Figure 4.5.

The vector execution time t_v and the parallel vector execution time t_{vp} are given by

$$\begin{aligned} t_v &= r_s \cdot t_s + r_v \cdot t_s / A_v \ , & (4.14) \\ t_{vp} &= r_s \cdot t_s + r_{vs} \cdot t_{vv} + r_{vp} \cdot t_{vv} / A_{vp} \ , & (4.15) \\ t_{vv} &= r_v \cdot t_s / A_v \ , & (4.16) \end{aligned}$$

where

A_v	= the vector acceleration factor, and
A_{vp}	= the parallel vector acceleration factor.

The total speedup S is given by

$$\begin{aligned} S &= \frac{t_s}{t_{vp}} & (4.17) \\ &= \frac{1}{r_s + (r_{vs} + r_{vp}/A_{vp}) \cdot r_v / A_v} & (4.18) \\ &= \frac{1}{r_s + r_{vs} \cdot r_v / A_v + r_{vs} \cdot r_v / (A_v \cdot A_{vp})} \ . & (4.19) \end{aligned}$$

Task $T = T_s + T_v; T_v = T_{vs} + T_{vp}$

T_s: Serial Portion

T_v: Vector Portion

T_{vs}: Serial Vector Portion

T_{vp}: Parallel Vector Portion

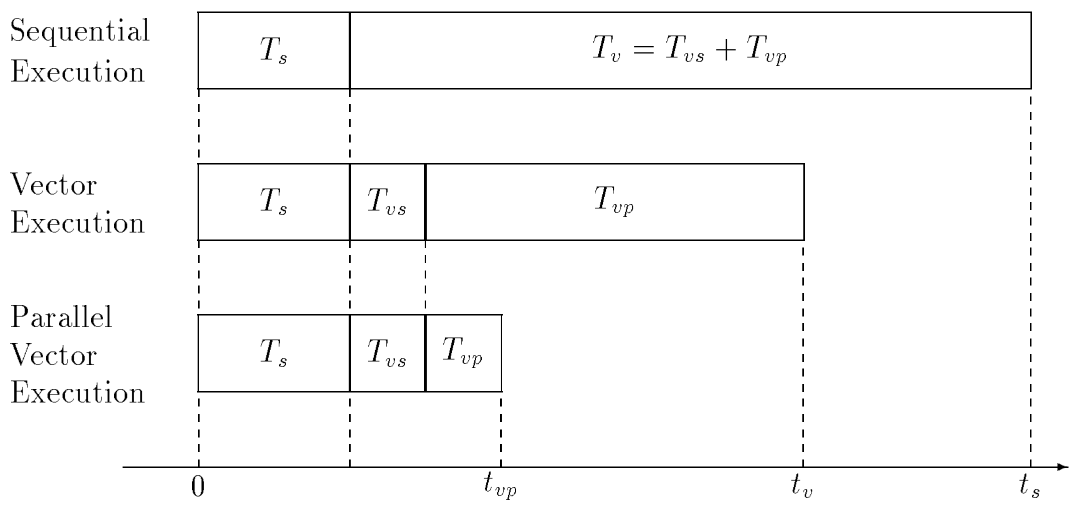


Figure 4.5: Sequential vs. Parallel Vector Execution.

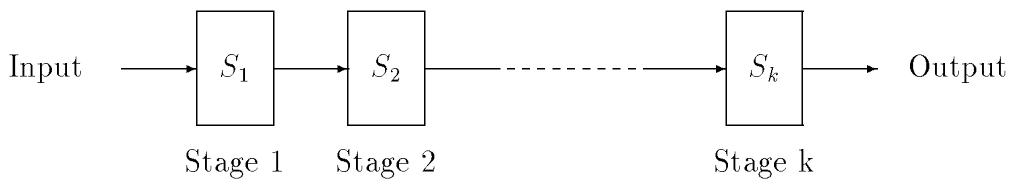


Figure 4.6: k-Stage Pipeline.

Pipeline Performance Model

We consider a uniform-delay pipeline, which is simple and widely used in many pipeline computing systems.

Figure 4.6 shows a k-stage uniform-delay pipeline. Let k, τ, and n be the number of pipeline stages, the clock period per stage, and the number of data, respectively.

Let T_k be the execution time of n data with a k-stage pipeline processor. It is given by

$$\begin{aligned} T_k &= k \cdot \tau + (n-1) \cdot \tau && (4.20) \\ &= (k-1) \cdot \tau + n \cdot \tau \ . && (4.21) \end{aligned}$$

where $(k-1) \cdot \tau$ is the startup time, and $n \cdot \tau$ is the apparent data processing time for n data items, each per τ. This is illustrated in Figure 4.7.

Let T_1 be the execution time of n data with the equivalent non-pipeline processor. It is given by

$$T_1 = n \cdot k \cdot \tau \ . \tag{4.22}$$

Then the speedup S_k of the k-stage pipeline processor over the equivalent non-pipeline processor is

$$S_k = \frac{T_1}{T_k} = \frac{n \cdot k \cdot \tau}{(k-1) \cdot \tau + n \cdot \tau} = \frac{n \cdot k}{k + n - 1} \ . \tag{4.23}$$

If $n \gg k$, that is, the number of data is very large compared to the number of pipeline stages, then

$$S_k \approx k \ . \tag{4.24}$$

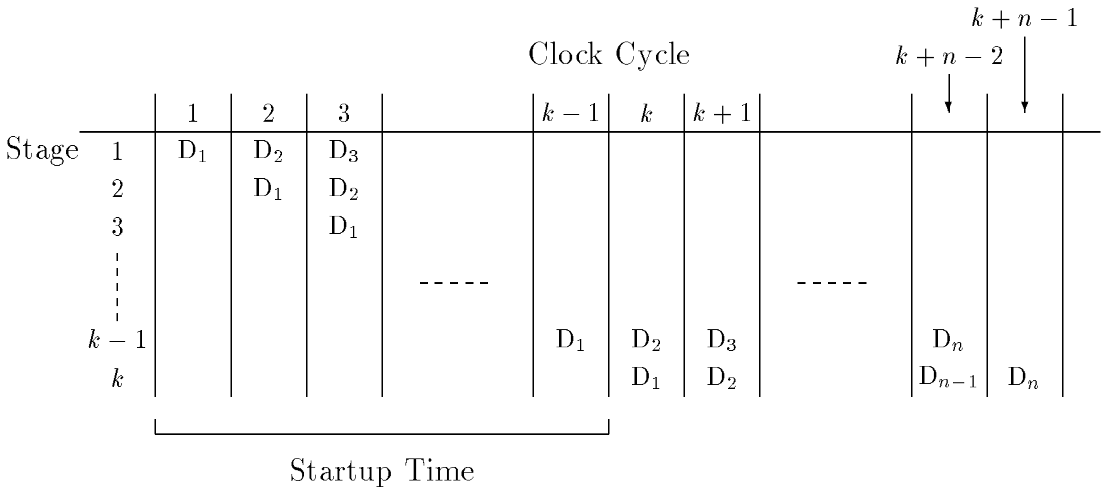


Figure 4.7: Data Processing Flow with k-Stage Pipeline.

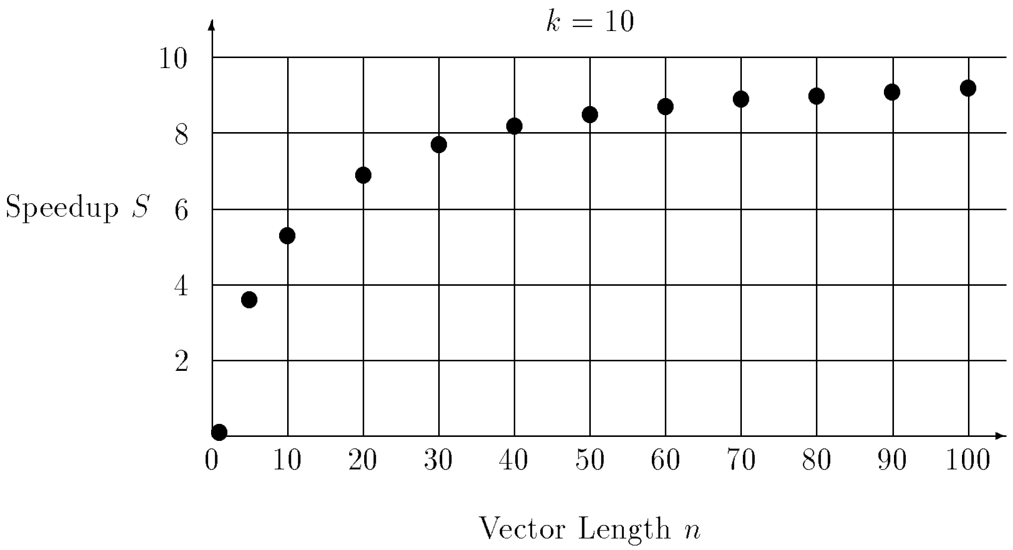


Figure 4.8: Speedup of Pipeline Prcessing.

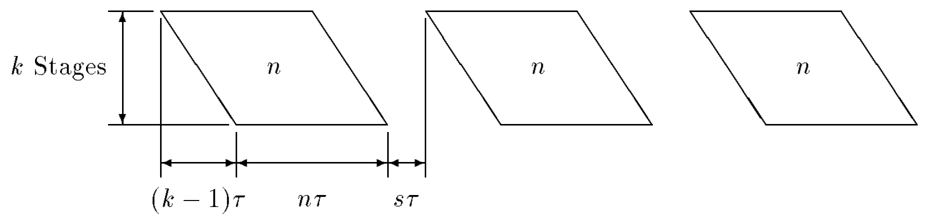


Figure 4.9: Pipeline Processing for Multiple Data Sets.

The maximum speedup with a k-stage pipeline processor is equal to the number of pipeline stages. Figure 4.8 shows the speedup of pipeline processing when n = 1, 5, 10, 20, . . . , 100 with $k = 10$.

We now consider a problem of processing multiple data sets in pipeline. Let n and m be the number of data items in a data set and the number of data sets, respectively. The total number of data item is given by

$$N = m \cdot n \; . \tag{4.25}$$

Assume that each data set is processed with a k-stage pipeline processor. Then the total processing time T'_k is given by

$$\begin{aligned} T'_k &= \{(k-1)\cdot\tau + n\cdot\tau\}\cdot m + (m-1)\cdot s\cdot\tau & (4.26)\\ &= \{(n+k+s-1)\cdot m - s\}\cdot\tau \; , & (4.27)\end{aligned}$$

where $s \cdot \tau$ is the setup time between one data set and the next. This is shown in Figure 4.9. The speedup S'_k is given by

$$\begin{aligned} S'_k &= \frac{T_1}{T'_k} & (4.28)\\ &= \frac{mnk}{(n+k+s-1)m - s} \; . & (4.29)\end{aligned}$$

If $m \gg s$, then

$$S'_k \approx \frac{nk}{n+k+s-1} \; . \tag{4.30}$$

If n is large compared to $k+s-1$, then S'_k is almost equal to k. If $s=-(k-1)$, then

$$S'_k = \frac{mnk}{nmk-s} \ . \qquad (4.31)$$

The setup time s has a negative effect on the total pipeline performance, although it doesn't affect the performance for processing each set of data. We may have several cases where data is partitioned into multiple data sets for processing. One of the cases is a natural case where data are sampled in groups at intervals. Another case is a situation that arises from some restrictions of the system. For example, if the system has a limited number of registers (e.g., vector registers), it has to process data that can be stored in the registers. In this case, the system has to store the computation results in the registers before loading the next set of data into the registers. The limited size of memories causes a similar situation. Yet another case is the situation where the application algorithm has to handle data sets separately.

We have investigated the performance of the pipeline through which data continuosly flow. There are some cases in which a pipeline can not be filled with data for some reasons. In these cases, we use dummy data to fill up the pipeline, rather than controlling clocking to the pipeline. Consider a k-stage pipeline that processes actual data and dummy data. Assume that data are organized in groups of k data items and that each group contains c actual data and $k-c$ dummy data ($0 \le c \le k$). Actual and dummy data alternately enter the pipeline; the pipeline processes c actual data and $k-c$ dummy data at any moment. Figure 4.10 illustrates the situation.

Now we consider the speedup for the case with actual and dummy data flowing alternately through a pipeline. Let T''_k be the execution time for n actual data with a k-stage pipeline. The pipeline processes $k-c$ dummy data for every c actual data; the total number of data is nk/c, instead of n. Then T''_k is given by

$$T''_k = \{(k-1)+nk/c\} \cdot \tau \ , \qquad (4.32)$$

where τ is the clock period per stage. The speedup S''_k is then given by

$$S''_k = \frac{T_1}{T''_k} = \frac{nk}{(k-1)+nk/c} \ . \qquad (4.33)$$

If $n \gg 1$, then

$$S''_k \approx c \ . \qquad (4.34)$$

This means that the speedup obtained with a k-stage pipeline is close to the number of actual data per group of k data items, not to the number of the pipeline stages k. That is, the speedup is bounded not by the number of stages but by the number of actual data when the pipeline alternately processes actual and dummy data.

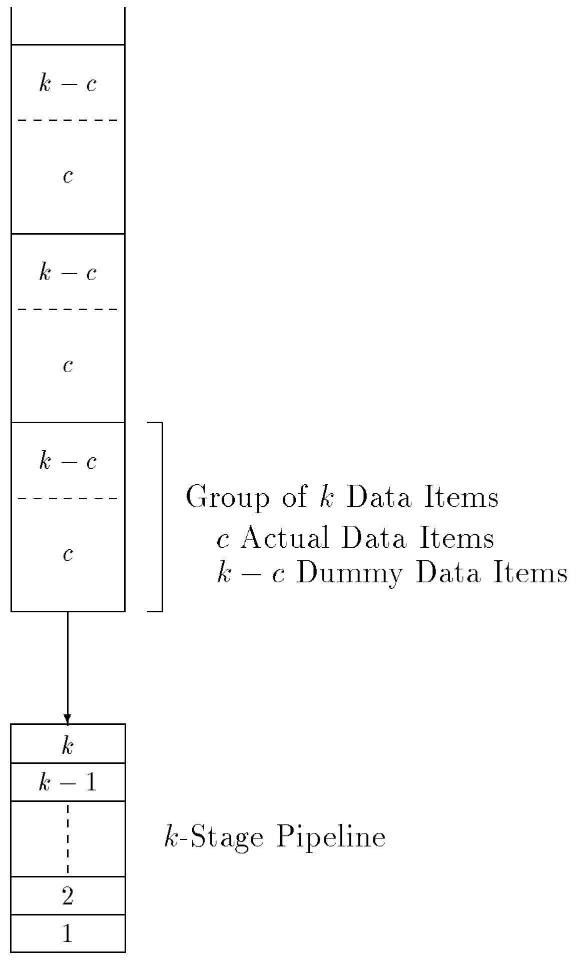


Figure 4.10: Pipeline Processing of Actual and Dummy Data.

4.3 Execution Model

Vector Loops

Many scientific and engineering applications, including real-time applications, involve vector computations, which are typically identified in loop constructs in high-level language programs. Not all of the loop constructs are in the vector form. The loop constructs that are amenable to vector processing are called *vector loops*. We distinguish the vector loops from other loops by using a **vector_for** construct.

Consider a simple vector-scalar multiplication as follows:

$$\mathbf{v} = \mathbf{w} \cdot s. \tag{4.35}$$

where $\mathbf{v}$ and $\mathbf{w}$ are n-element vectors and s is a scalar. Each element of the vector $\mathbf{v}$ can be computed as follows:

$$v_i = w_i \cdot s \quad \text{for} \quad i = 1, 2, \ldots, n. \tag{4.36}$$

This computation can be programmed using a C-like language as shown below:

```
for ( $i = 1$; $i <= n$; $i++$ )
{
    $v[i] = w[i] * s$;
}
```

Since it is in the vector form, it can be expressed with a **vector_for** loop as follows:

```
vector_for ( $i = 1$; $i <= n$; $i++$ )
{
    $v[i] = w[i] * s$;
}
```

Consider another example of a matrix multiplication given by

$$\mathbf{A} = \mathbf{B} \times \mathbf{C}, \tag{4.37}$$

where $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are all matrices of $n \times n$. Each element of $\mathbf{A}$ can be computed as follows:

$$a_{ij} = \sum_{k=1}^{n} b_{ik} \cdot c_{kj}. \tag{4.38}$$

This matrix multiplication can be programmed with three nesting **for** loops. Since the vector loop is limited to the innermost loop, the program with the **vector_for** loop is described as follows:

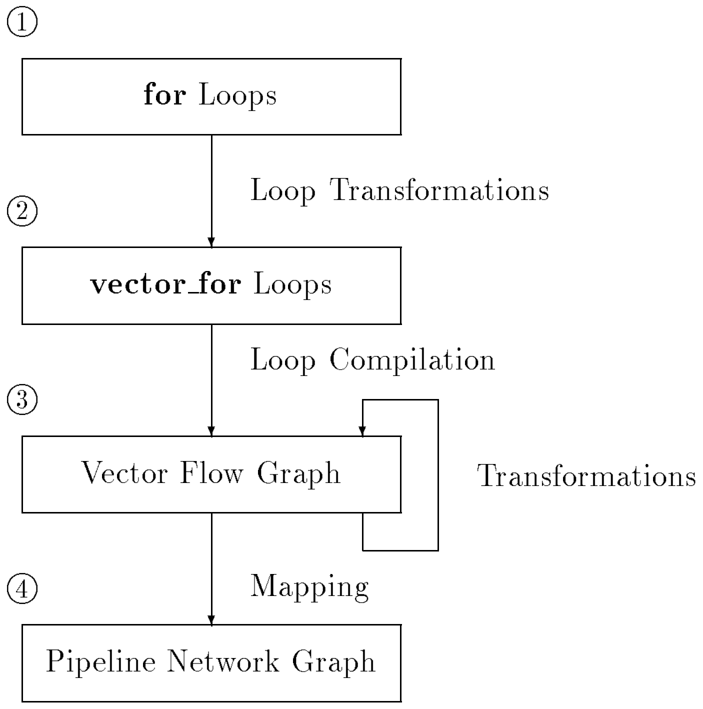
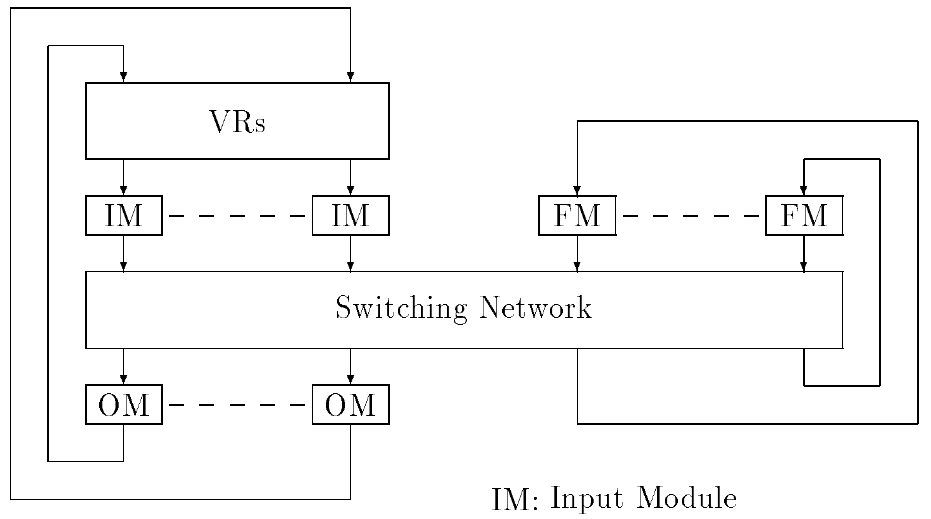


Figure 4.11: A Pipeline Network Generation Flow.

```
for ( $i = 1; i <= n; i++$ )
{
    for ( $j = 1; j <= n; j++$ )
    {
        $a[i][j] = 0;$
        vector_for ( $k = 1; k <= n; k++$ )
        {
            $a[i][j] = a[i][j] + b[i][k] * c[k][j];$
        }
    }
}
```

We are concerned with the **vector_for** loops. In the following sections, we present the



IM: Input Module
OM: Output Module
FM: Functional Module
VRs: Vector Registers

Figure 4.12: A Logical Machine Model.

techniques to generate a pipeline network from a **vector_for** loop. The procedural flow for generating a pipeline network from a **for** loop is shown in Figure 4.11. The flow starts with a given **for** loop in the first box. Through some loop transformations, which are beyond the scope of this thesis, **for** loops are transformed to **vector_for** loops in the second box. Then they are represented by vector flow graphs in the third box, using the compilation algorithms. The vector flow graphs are then converted to pipeline network graphs in the fourth box through several transformations (pipeline chaining, delay insertion, etc.). The pipeline network graph is an abstract network graph for a particular logical pipeline network configuration.

The Logical Vector Machine Model

We define a logical vector machine model that executes the **vector_for** loops in pipeline. Its general structure is illustrated in Figure 4.12. In this figure, VRs are vector registers that hold input, output and intermediate vector data, IM is an input module that reads vector data from VRs, OM is an output module that writes vector data into VRs, and FM is a functional module that performs some arithmetic on vector data. These modules are interconnected via a switching network. We assume that all the vector data to be processed are stored in VRs and all the result vector data are also stored in VRs.

An FM is either a single-cycle module or a multi-cycle module. A single-cycle module takes one pipeline cycle in processing vector data; a multi-cycle module is a pipeline module and takes multiple cycles in processing vector data. If all the FMs are single-cycle modules, the interconnections of these FMs through the switching network form one or more pipelines. If multi-cycle modules are included, then their interconnections through the switching network form two-level pipeline networks.

The switching network can be a set of data buses, a crossbar switch, or a multi-stage interconnection network. For simplicity, we ignore the delay of the switching network and assume no restrictions on the interconnections of modules. Here we assume that there are an arbitrary number of data buses available for connections, numbered B_1, B_2, B_3, ..., in the switching network. We also assume that the CONNECT command is available to connect one module to the specified data bus. For example, consider the following three CONNECT commands.

CONNECT ALU_5.S B_3;
CONNECT ALU_5.A B_2;
CONNECT ALU_5.B B_7;

The first CONNECT command connects the output S of the functional module ALU_5 to the data bus B_3, the second CONNECT command connects the input A of ALU_5 to B_2, and the third CONNECT command connects the input B of ALU_5 to B_7. Using the CONNECT command, we can "program" the switching network to form logical connections between modules for a particular operation. We assume that the connections made by the CONNECT commands stay valid until the DISCONNECT command is used to disconnect them. All the modules are disconnected from the data buses by issuing

DISCONNECT ALL;

and one module is disconnected from the data buses by

DISCONNECT ALU_5;

The last command disconnects all the inputs (A and B) and output (S) of the module ALU_5 from the data buses.

Vector Flow Graph

A *vector flow graph* is a graphical representation of a **vector_for** loop and is a special form of a *flow graph*.

A *flow graph* $G = (V, E)$ is a directed graph where

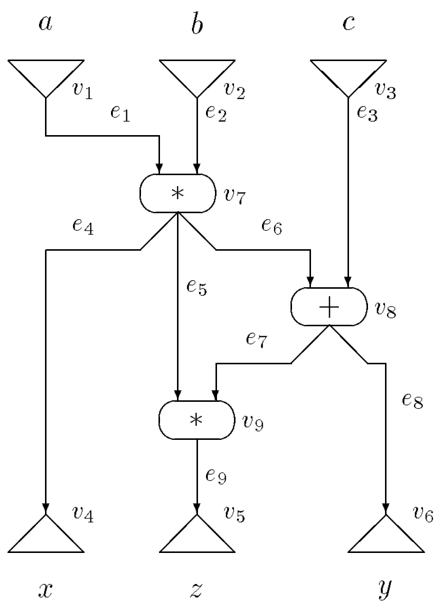


Figure 4.13: An Example Flow Graph.

$V = \{v_1, v_2, \ldots, v_n\}$: a set of nodes; and
$E = \{e_1, e_2, \ldots, e_m\}$: a set of edges.

A node v_i is either an operation node, an input node, or an output node. The input and output nodes are distinguished from the operation nodes due to their special roles.

A flow graph is produced from a program segment written in a C-like high-level language. Consider the following program segment:

$x = a * b;$
$y = x + c;$
$y = x * y;$

where the variables a, b, c, x, y, and z are all scalars. The corresponding flow graph is shown in Figure 4.13. This flow graph is represented by $G = (V, E)$, where

$V = \{v_1, \ldots, v_9\}$; and
$E = \{e_1, \ldots, e_9\}$.

In Figure 4.13, v_1, v_2, and v_3 are input nodes (for a, b, and c, respectively), v_4, v_5, and v_6 are output nodes (for x, y, and z, respectively), and v_7, v_8, and v_9 are operation nodes (for $*$, $+$, and $*$, respectively).

A flow graph produced from the body of a **vector_for** loop is called the *vector flow graph*. For example, consider the following program segment with a **vector_for** loop:

```
vector_for ( i = 1; i <= 100; i++ )
{
    x[i] = a[i] * b[i] + c[i];
}
```

The vector flow graph for the body of this **vector_for** loop is shown in Figure 4.18.

We assume that the body of the **vector_for** loop consists of a sequence of assignment statements, whose lefthand side is a vector element. We call these assignment statements *vector assignment statements*. A vector flow graph is created from each vector assignment statement. Since one assignment statement has only one output, the vector flow graph produced from it is a tree. We call it a *vector assignment tree*. The following algorithm creates a vector assignment tree for a given assignment statement.

[Vector Assignment Tree Generation Algorithm]

Input: A vector assignment statement S
Output: A vector assignment tree T

Procedure:

Step 1: Create an output node v_{out} for the vector element on the lefthand side.
Step 2: Create an input nodes v_{in_j} for each unique variable on the righthand side.
Step 3: Create an operation node v_i for each operator on the righthand side.
Step 4: Create an edge e_k from an operator to its operand.

The vector assignment tree is basically a syntax tree. It is easily created from the postfix form of the righthand side of the vector assignment statement. Scanning the postfix form from left to right, create an input node if the token is a variable and it is unique, or create an operation node if the token is an operator. When an operator node is created, create the edge(s) between the operator and the previous one or two variables, depending on the type of the operator.

The body of a **vector_for** loop may consist of more than one vector assignment statement. If two consecutive vector assignments have some variables in common, two separately generated vector assignment trees can be merged. If the output vector element of a vector assignment statement appears on the righthand side of the subsequent vector assignment statement, Then the merging operation is called *chaining*. If the two assignment statements have no common variables, then no effective merging operation takes place. This procedure is described as follows:

[Vector Assignment Tree Merge Algorithm]

Input: two vector assignment trees T_1 and T_2
Output: a vector flow graph G

Procedure:

Step 1: If there are common input nodes in T_1 and T_2, then eliminate one of them and connect all the edges connected to it to the other one.

Step 2: If there is an input node in T_2 that has the same label (the associated variable) the output node of T_1 has, then eliminate that node and connect all the outgoing edges from the node to the output node.

Step 3: Create a graph with all the nodes and edges.

Figure 4.14 shows an example of the tree merge, where trees T_1 and T_2 are merged. The output node of T_1 is connected to an input node of T_2 (both of them denote the identical data $x[i]$). Since T_1 and T_2 have the same input $a[i]$, the input of T_2 is eliminated and an edge from the input of T_1 is added.

This algorithm can be applied successively to all the vector assignment statements in a **vector_for** loop, producing a vector flow graph. Now we describe the procedure that creates a vector flow graph from a **vector_for** loop.

[Vector Loop Compilation Algorithm]

Input: the body of a **vector_for** loop L
Output: a vector flow graph G

Procedure:

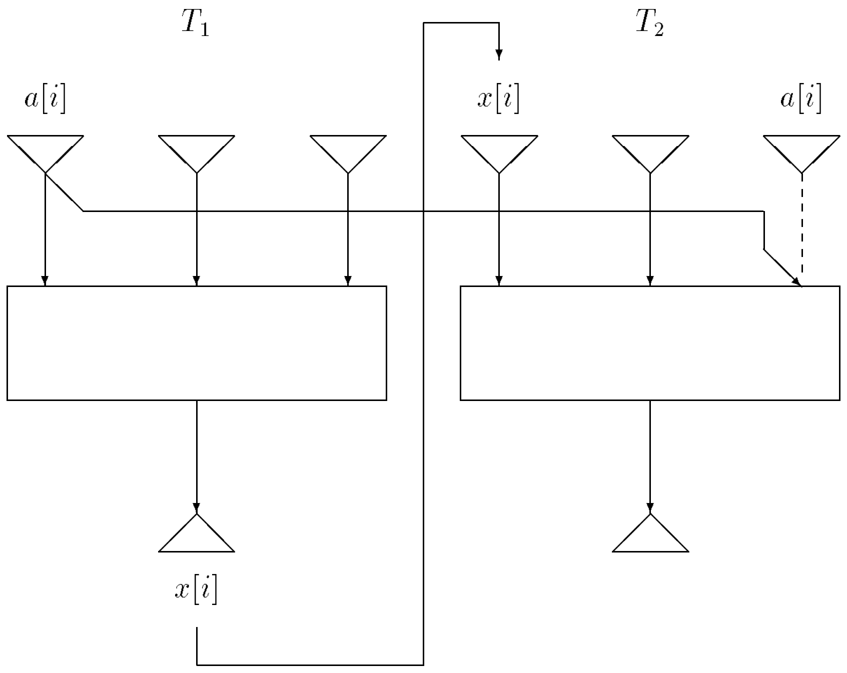


Figure 4.14: Vector Assignment Tree Merge.

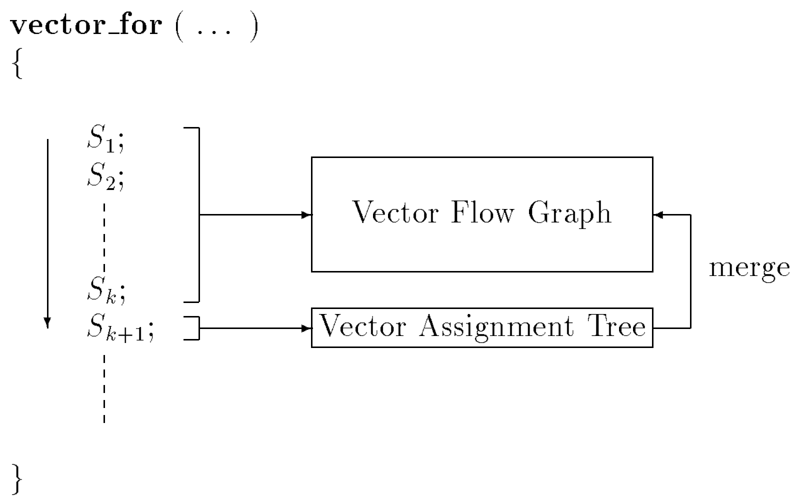


Figure 4.15: Constructing a Vector Flow Graph.

Step 1: Create a vector assignment tree for each of the vector assignment statement in L using the vector assignment tree generation algorithm.

Step 2: Create a vector flow graph G by applying the vector assignment tree merge algorithm successively to the vector assignment trees created in Step 1.

Figure 4.15 illustrates this process, where the vector assignment tree created from the statement S_{k+1} is merged into the vector flow graph created from the statements S_1, S_2, $\ldots$, S_k.

Pipeline Network Graph

A *pipeline network graph* is an abstract network graph that reflects the logical configuration of a machine model. It can be directly implemented on the logical vector machine model. Figure 4.16 shows an example pipeline network graph. Figure 4.17 is a simplified graph. In these graphs, FADD is a pipeline floating-point adder module with 5 pipeline stages, FMUL is a pipeline floating-point multiplier module with 4 pipeline stages, and PDLY is a programmable delay module programmed for 4 stages. Both of the graphs represent the pipeline network for the following computations:

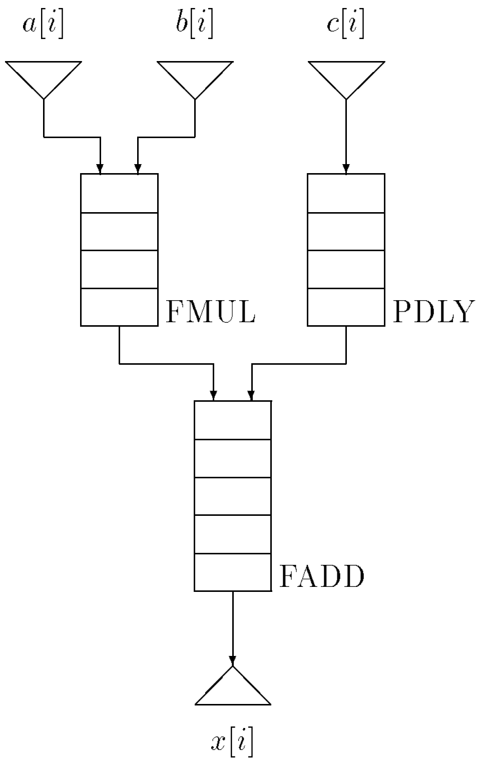


Figure 4.16: An Example Pipeline Network Graph.

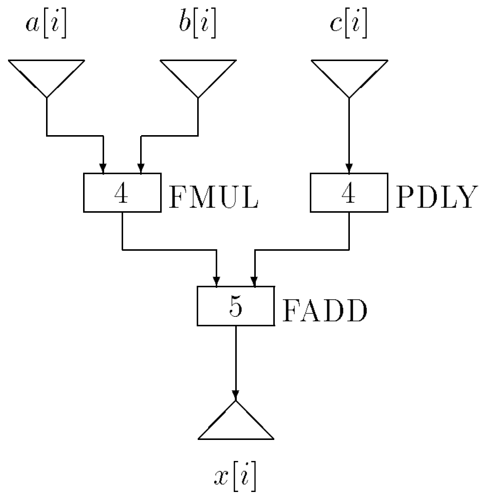


Figure 4.17: A Simplified Pipeline Network Graph.

```
vector_for ( $i = 1; i <= 100; i++$ )
{
    $x[i] = a[i] * b[i] + c[i];$
}
```

The vector flow graph generated from this **vector_for** loop is shown in Figure 4.18.

The difference between the vector flow graph and the pipeline network graph is the existence of the delay module in the pipeline network. The vector flow graph is a syntax tree of the statements in the **vector_for** loop and represents the dependencies of data, whereas the pipeline network graph represents the data flows through the pipeline. In order to obtain the pipeline network graph for a given **vector_for** loop or its vector flow graph, a procedure is required to insert the delays to synchronize the data flows. The synchronous data flow is essential to pipeline processing. In order to keep the consistency in pipeline processing, all the paths from an input to an output must include the same number of pipeline stages. Delay insertion is required to make all the data go through the same number of pipeline stages.

Delay Insertion

Delay insertion is a transformation on a vector flow graph to obtain a pipeline network graph that is directly implementable on the logical vector machine model.

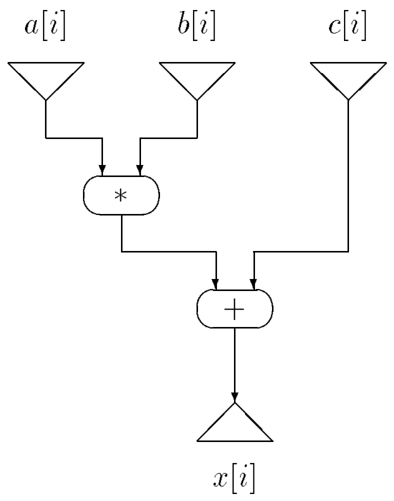


Figure 4.18: A Vector Flow Graph.

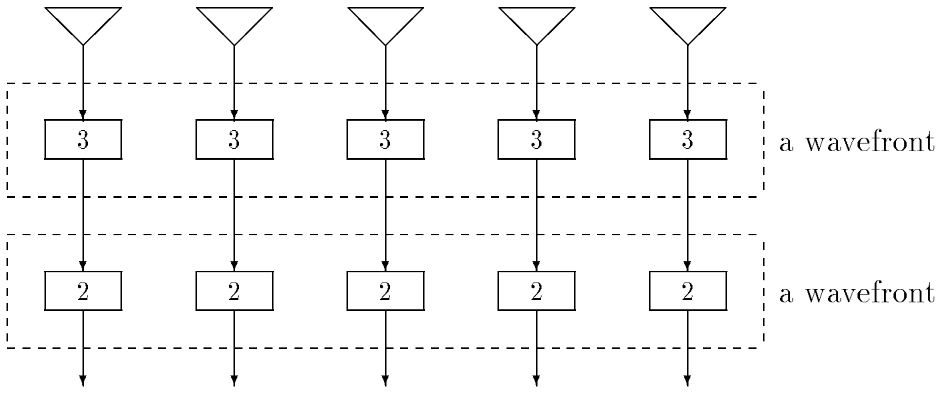


Figure 4.19: Wavefronts in Pipeline.

A set of data flowing through the pipeline network at the same time is called a *wavefront*. A wavefront consists of data (vector elements) that came out of input nodes at the same time. We can define the validity of the pipeline network using the wavefront. Assume that all the data of a wavefront are in input modules at cycle 0. We say that a wavefront is at the (pipeline) stage k if all the data of the wavefront are at the stage k. The pipeline network is *valid* if a wavefront is at the stage k at cycle k; otherwise, it is *invalid*. If the pipeline network is valid, then data in a wavefront reach the output nodes at the same time. Figure 4.19 shows wavefronts.

Coming out of the input nodes, a wavefront reaches the operation nodes connected from the input nodes through their outgoing edges. Then it propagates through the operation nodes and go to the next operation nodes through the connected edges. Since different operation nodes may have different delays, some data of the wavefront may come out of the operation nodes and the others stay at the operation nodes. As long as the pipeline network is valid, however, the wavefront should be at the stage k at cycle k.

The following delay insertion algorithm makes the pipeline network valid.

[Delay Insertion Algorithm]

Input: a vector flow graph G
Output: a pipeline network graph P

Procedure:

Step 1: Assign the delay to each operation node in G. The delay is the number of stages in the pipeline formed in the operation node.

Step 2: From each input node, follow all the edges and nodes on the path(s) to the output module(s) in the breadth-first fashion. Assign 0 to all the outgoing edges of all the input nodes. Follow the outgoing edges of the input nodes.

Let $N_{in}(i)$ be the number of incoming edges of the operation node v_i. Similarly, let $N_{out}(i)$ be the number of outgoing edges of the operation node v_i.

Step 2a: At each operation node v_i:

Step 2a-1: If $N_{in} = 1$, then $A(i) =$ the value assigned to the incoming edge.

Step 2a-2: If $N_{in} > 1$, then $A(i) =$ the max of the values assigned to the incoming edges.

Step 2a-3: If $N_{in} > 1$, for each edge whose assigned value (x) is less than $A(i)$, insert the delay node of $A(i) - x$ in the middle of the edge.

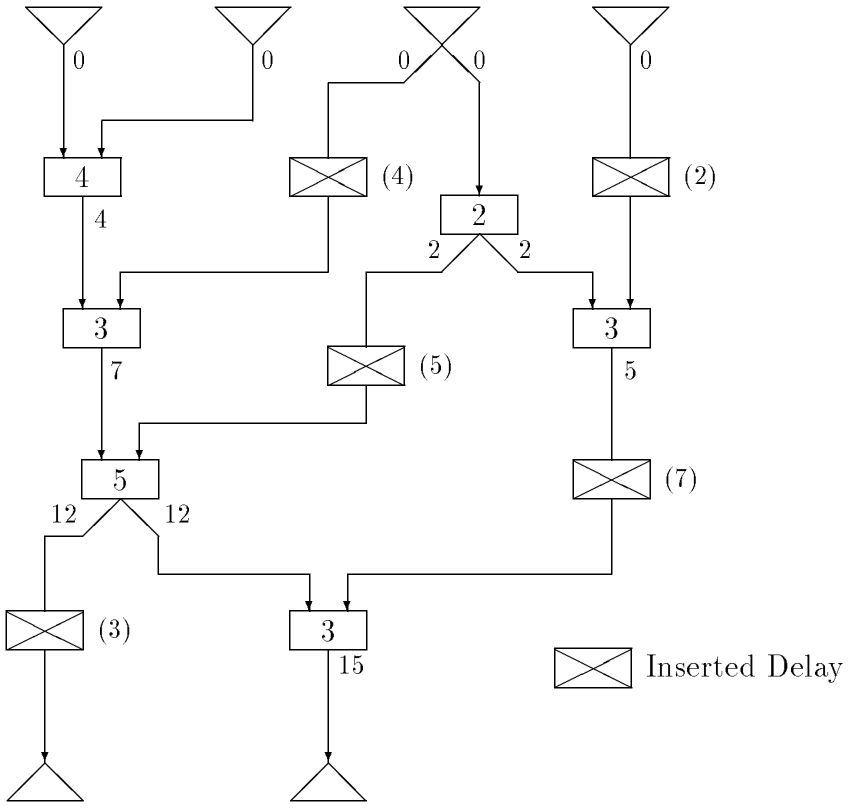


Figure 4.20: An Example of Delay Insertion.

Step 2b: At each outgoing edge of the operation node v_i, assign $d + A(i)$ to it, where d is the delay of the operation node v_i.

Step 2c: Follow each outgoing edge of the operation node v_i and repeat Step (2a) until it reaches the output node.

Step 3: Compare all the values assigned to the edges going to the output nodes and take the maximum as M. Then insert the delay of $M - v$ on each of the edges with the value v.

Figure 4.20 shows an example of delay insertion, where five delays are inserted.

In the above-mentioned algorithm, a delay is inserted so that each operation node receives its operands in the same wavefront. Since all the operation nodes receive their operands in the same wavefront in the transformed graph, it is a valid pipeline network.

Vector Reductions

A general vector-reduction function g is defined by

$$g : V_1 \times V_2 \times \cdots \times V_k \rightarrow S, \tag{4.39}$$

where g is a k-ary function, V_i for $i = 1, 2, \ldots, k$ are vectors, and S is a scalar. For instance, the inner product is a binary vector-reduction function ($k = 2$). Here we consider only the unary vector-reduction function f, that is,

$$f : V \rightarrow S. \tag{4.40}$$

Let v be a vector with n elements. Consider the vector-reduction of the form:

$$x = v_1 \diamond v_2 \diamond \cdots \diamond v_n, \tag{4.41}$$

where $\diamond$ is a binary operator defined by f. It can be programmed as follows:

```
$x = v[1];$
**for** ( $i = 1; i < N; i++$ )
{
        $x = x \diamond v[i];$
}
```

where x is a scalar variable.

If the operation module for $\diamond$ has k pipeline stages in it, then the program can be rewritten as follows:

```
$x[0] = x[-1] = \cdots = x[-(k-1)] = a;$
**vector_for** ( $i = 1; i < N; i++$ )
{
        $x[i] = x[i-k] \diamond v[i];$
}
$z = x[N-1] \diamond x[N-2] \diamond \cdots \diamond x[N-k];$
```

where $a \diamond x = x$ for any x. In this program, $x[N-j]$ for $1 \leq k$ is given by

$$x[N-j] = v[N-j] \diamond v[N-j-k] \diamond v[N-j-2k] \diamond \cdots . \tag{4.42}$$

Because of the k pipeline stages in the operation module, the last k elements of the vector x have to be added up with $\diamond$ after the **vector_for** loop. This **vector_for** loop can be implemented with a feedback path as shown in Figure 4.21.

If there are k similar statements in the **vector_for** loop, then

```
$x_1 = v_1[1];$
    :
$x_k = v_k[1];$
for ( $i = 1; i < N; i++$ )
{
        $x_1 = x_1 \diamond v_1[i];$
            :
        $x_k = x_k \diamond v_k[i];$
}
```

In this case, k independent statements are alternately processed in the same pipeline. If there are fewer indenpendent statemenst, we can add dummy statements. Then the performance of the pipeline is bounded by the number of actual data items per group of k data items, as discussed in the previous chapter.

Linear Recurrences

An m-th order linear recurrence system of n equations $R < n, m >$ is defined for $m \leq n-1$ by

$$x_i = 0 \quad \text{for} \quad i \leq 0, \tag{4.43}$$

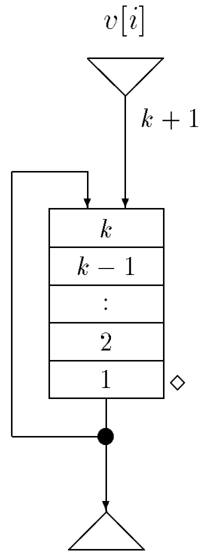
$$x_i = c_i + \sum_{j=i-m}^{i-1} a_{ij} x_j \quad \text{for} \quad 1 \leq i \leq n. \tag{4.44}$$

It has the form

$$x_i = c_i + a_{i,i-m} x_{i-m} + a_{i,i-m+1} x_{i-m+1} + \cdots + a_{i,i-1} x_{i-1} \quad \text{for} \quad i = 1, 2, \ldots, n. \tag{4.45}$$

For example, an $R < 5, 2 >$ system has the form

$$x_1 = c_1, \tag{4.46}$$



$x[i] = x[i-k] \diamond v[i]$
Figure 4.21: A Pipeline Network for Vector Reduction.

$$x_2 = c_2 + a_{21}x_1, \tag{4.47}$$
$$x_3 = c_3 + a_{31}x_1 + a_{32}x_2, \tag{4.48}$$
$$x_4 = c_4 + a_{42}x_2 + a_{43}x_3, \tag{4.49}$$
$$x_5 = c_5 + a_{53}x_3 + a_{54}x_4. \tag{4.50}$$

If $m = n-1$, the system is called a general linear recurrence system and denoted by $R < n >$.

Consider the following program segment

```
/* $c > 0$ */
**for** ( $i = c + 1; i < N; i++$ )
{
    $x[i] = a[i] \diamond x[i-c]$;
}
```

where $\diamond$ denotes some arithmetic operation. In this assignment statement, the i-th element of the vector x is computed using the $(i-c)$-th element of the same vector. Because of this dependency, $x[i]$ can not be computed until $x[i-c]$ is computed. We assume that the operation module for the operation $\diamond$ has k pipeline stages in it.

If $k < c$, then the pipeline network can be configured with a feedback path and a delay. For example, consider

```
**vector_for** ( $i = 6; i < 100; i++$ )
{
    $x[i] = a[i] \diamond x[i-5]$;
}
```

If the operation module that performs $\diamond$ has 3 pipeline stages in it, it can be computed with the pipeline network graph shown in Figure 4.22, which shows a snap shot at $i = 15$.

In the case of $k < c$, we can configure the pipeline network without a feedback path as shown in Figure 4.23. Since $k < c$, $x[i]$ can be computed with $a[i]$ and $x[i-c]$, both read from the VRs, and stored into the VRs.

If $k \geq c$, then we need to rewrite the program by rewriting the vector assignment statement. $x[i]$ can be expressed as follows:

$x[i] = a[i] \diamond a[i-c] \diamond x[i-2c]$;
$x[i] = a[i] \diamond a[i-c] \diamond a[i-2c] \diamond x[i-3c]$;
...
$x[i] = a[i] \diamond \cdots \diamond a[i-(t-1)c] \diamond x[i-tc]$;
...

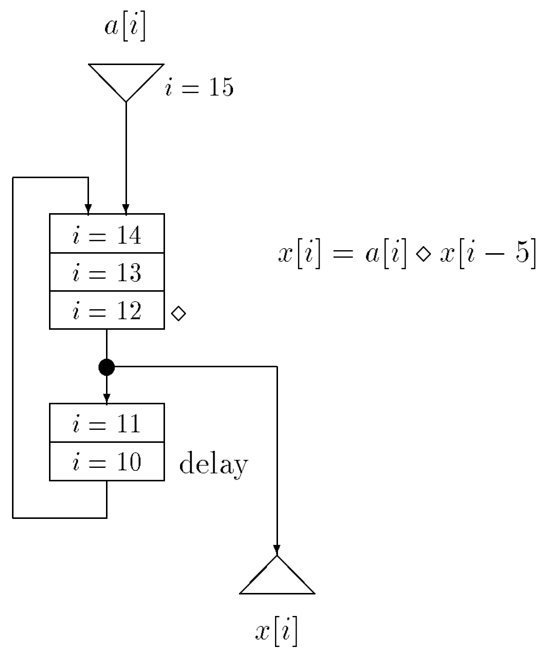


Figure 4.22: A Pipeline Network for Recurrence.

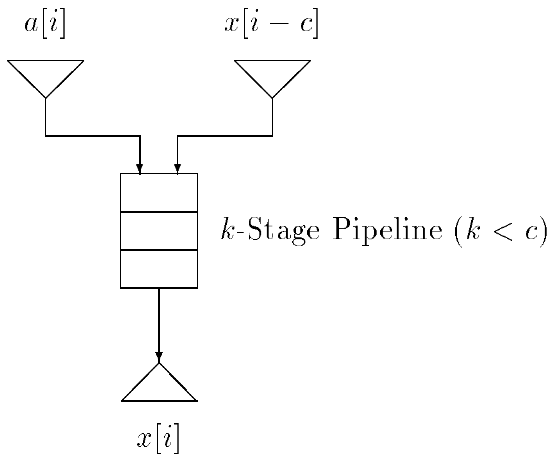


Figure 4.23: Another Pipeline Network for Recurrence.

Choose the smallest value of t such that $k < tc$. Then $x[c+1]$, $x[c+2]$, ..., $x[tc]$ have to be precomputed before the **vector_for** loop. As an example, consider

```
for ( $i = 4$; $i < 100$; $i++$ )
{
$x[i] = a[i] \diamond x[i-3]$;
}
```

If the operation module for $\diamond$ has 7 pipeline stages in it, then the loop is rewritten as follows:

```
$x[4] = a[4] \diamond x[1]$;
$x[5] = a[5] \diamond x[2]$;
...
$x[9] = a[9] \diamond x[6]$;
vector_for ( $i = 10$; $i < 100$; $i++$ )
{
$x[i] = a'[i] \diamond x[i-9]$;
}
```

where $a'[i]$ is assumed to be precomputed by

$$a'[i] = a[i] \diamond a[i-3] \diamond a[i-9].$$

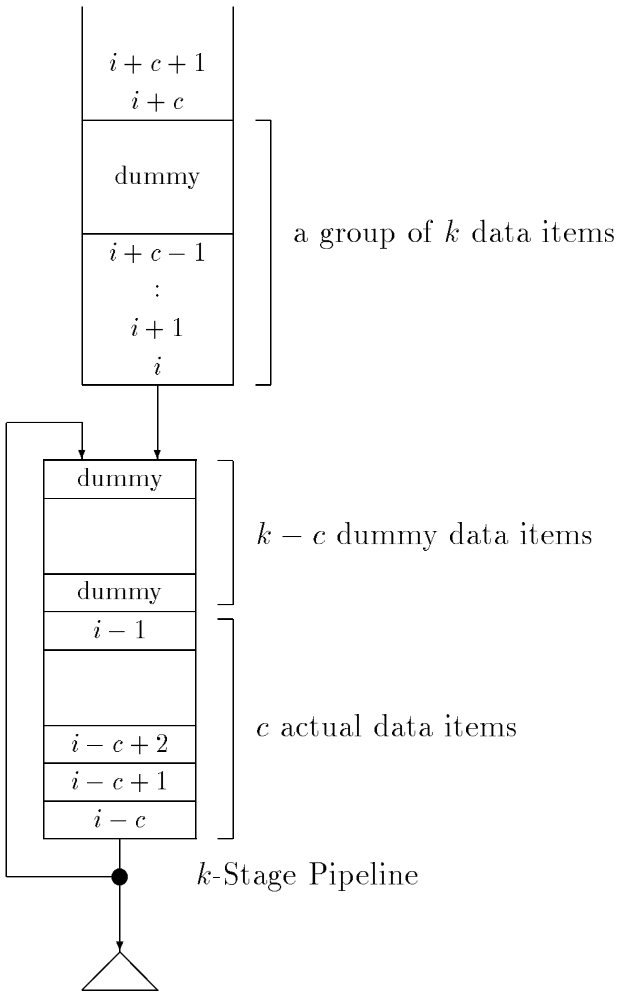


Figure 4.24: A Pipeline Network with Dummy Data.

Then we can apply the method for $k < c$ to the transformed **vector_for** loop.

Another way to handle the case of $k \geq c$ is to insert $k - c$ dummy statements in the **vector_for** loop so that $x[i]$ can be computed with $x[i-c]$ after $x[i-c]$ has been computed. This method requires groups of k data items, including c actual data items and $k-c$ dummy data items. In processing each group of k data items, $x[i]$, $x[i+1]$, $x[i+2]$, $\ldots$, $x[i+c-1]$ are computed. This is illustrated in Figure 4.24. As discussed in the previous chapter, the pipeline performance is bounded by $c < k$ in this case.

Chapter 5

DRA System Architecture

In this chapter, we propose a computing system architecture, called the *Dynamically Reconfigurable Architecture* or *DRA*, for the target class of real-time computing. We focus on the major architectural features.

5.1 Overall System Structure

The DRA system consists of the following system components:

- Scalar Processor Unit (SPU)
- Vector Processor Unit (VPU)
- Global Memory Unit (GMU)
- Input/Output Units (IOUs)

The SPU is an autonomous processor unit that performs scalar operations and system control operations. The VPU is a semi-autonomous processor unit that performs vector operations under the control of the SPU. The GMU is a memory unit that stores input, intermediate, and output data. The IOUs are the units for input and output operations.

As shown in Figure 5.1, these system components are interconnected via four system buses. Two of them are used to transfer data between the GMU and IOUs; the other two buses are used to transfer data between the GMU and the VPU. Four system buses allow the system to perform input/output operations and vector operations at the same time. Each bus can be used for any type of data transfer operation. In one of the typical system activities, the following data transfer operations might be performed in parallel:

- IOU to GMU: real-time data input
- GMU to IOU: real-time data display
- GMU to VPU: inputs for vector computations
- VPU to GMU: results of vector computations

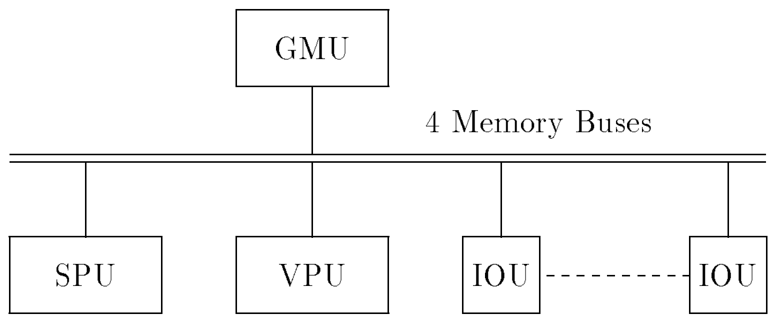


Figure 5.1: DRA System Structure.

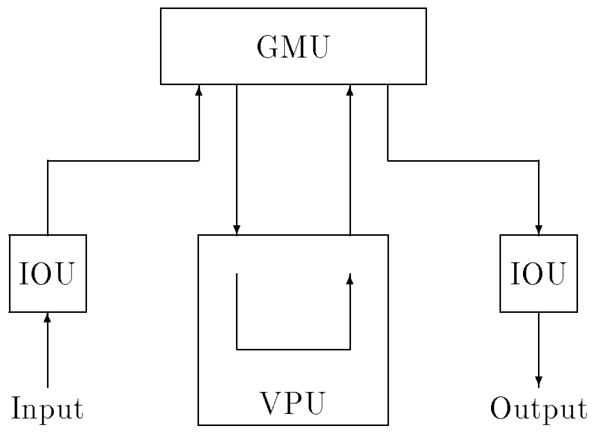


Figure 5.2: DRA System-Wide Pipelines.

The capability of these four parallel data transfer operations enables the system to operate globally in pipeline. This system-wide pipeline consists of five stages: the first stage for real-time data input, the second for vector data input, the third for vector computations, the fourth for vector data output, and the fifth for result data output, as shown in Figure 5.2.

In general, the time spent for each data transfer operation is proportional to the number of data (input real-time data); that is, $O(n)$, where n is the number of data. However, the time spent for vector computations is not necessarily in proportion to n. Although it depends on the algorithms and the available resources, in many complex vector computations, it is $O(n \log n)$ or $O(n^2)$, which is larger than $O(n)$. In such a case, data transfer operations terminate before vector computations finish, if they start at the same time.

When we design a DRA system for a specific real-time application in the target class, the time period of this system-wide pipeline is upper-bounded by the cycle time of periodic real-time data input, which is the maximum time allowed for data transfer operations and vector computations at the pipeline stages.

5.2 Scalar Processor Unit (SPU)

The Scalar Processor Unit or SPU performs scalar operations and controls the other system components. It consists of the Scalar Memory Unit (SMU) and the Scalar Execution Unit (SXU), as shown in Figure 5.3. The SXU is an autonomous processor that executes programs stored in the SMU. It reads machine instructions from the SMU and executes them. Data referenced by an instruction are also fetched from the SMU and the results of the instruction execution are stored back to it. Data can be moved back and forth between the SMU and the GMU by a DMA-like operation.

The SPU controls the other components through its control interface (control address and control data) as shown in Figure 5.3. The control address is the address of the location to access, and the control data is the data to send to the location or to receive from the location. Control addresses are uniquely assigned to the various locations for control in the system components. This interface is used to initiate vector operations in the VPU. As illustrated in Figure 5.4, in a typical case, the SPU first prepares for a vector operation (V_i) in the VPU (C_i) and then initiates the VPU using the control interface. After the VPU starts performing the vector operation, the SPU does its own scalar task (S). When the VPU terminates the vector operation, it signals the termination to the SPU by INT (interruption). The SPU prepares for the next vector operation (V_{i+1}) and repeats the process.

The SPU can be a single-processor unit that contains a CPU (SXU) and a main memory (SMU). It requires nothing special for the DRA and can be built with a commercial (RISC) microprocessor.

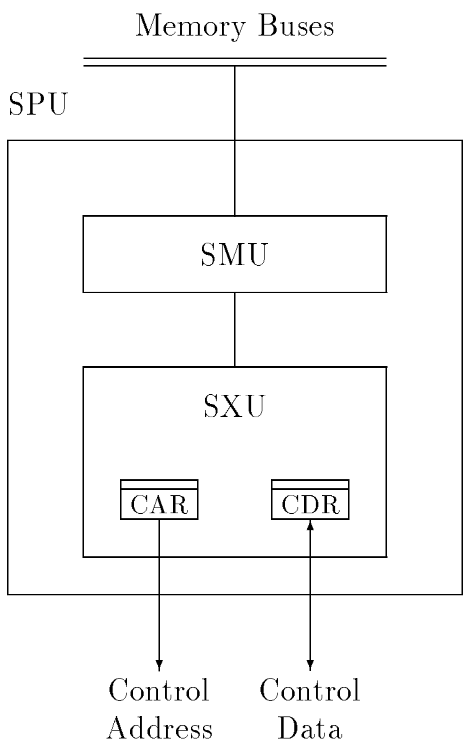
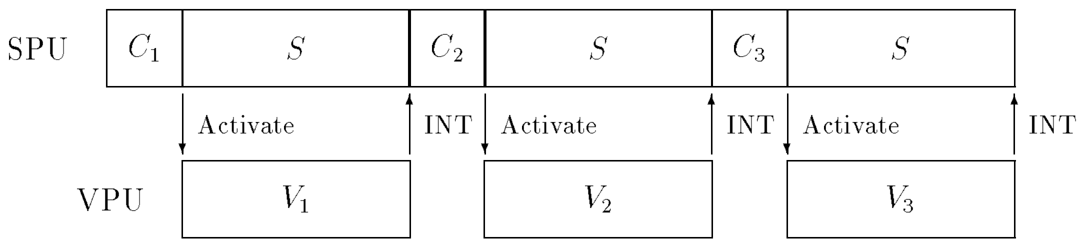


Figure 5.3: SPU Structure.



S: Scalar Operation
C_i: Control Operation for V_i
V_i: Vector Operation
INT: Interrupt

Figure 5.4: SPU and VPU Operations.

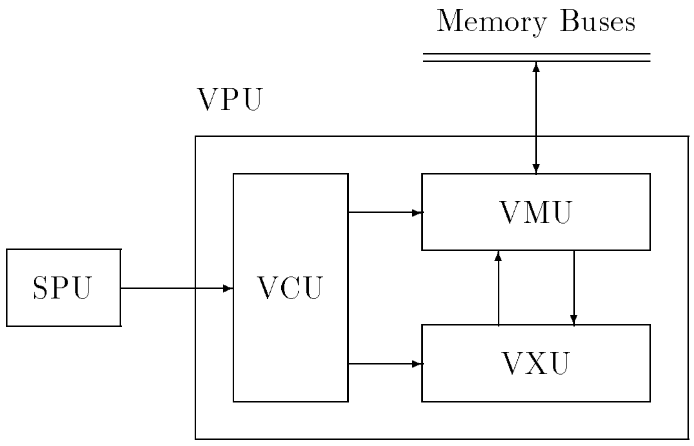


Figure 5.5: VPU Structure.

5.3 Vector Processor Unit (VPU)

The Vector Processor Unit or VPU performs vector operations in pipeline under the control of the SPU. It consists of the Vector Memory Unit (VMU), the Vector Execution Unit (VXU), and the Vector Control Unit (VCU), as shown in Figure 5.5.

The VMU stores vector data, input, intermediate, and output. As shown in Figure 5.6, it consists of four vector memory blocks so that four types of memory access can be performed in parallel. Each memory block can process memory access requests independently. In Figure 5.6, VMB_0 is used for write from one of the memory buses, VMB_1 for read to one of the memory buses, VMB_2 for read to the VXU, and VMB_3 for write from the VXU. These four types of memory access activities may continue, but the memory blocks used for each memory access type may change. For example, at the next stage after the vector operation terminates in the VXU, VMB_0 may be used for read to the VXU, VMB_1 for write from the VXU, VMB_2 for write from a memory bus, and VMB_3 for read to a memory bus in the next stage.

As shown in Figure 5.7, the VCU contains a control register (CR) that holds a very long instruction to control the vector operations in the VXU. A sequence of instructions is read into the control register from the control memory (CM) and sets up for a vector operation to be performed in the VXU. The instruction sequence is controlled by the sequencer (SEQ). When the instruction sequence terminates with a halt operation, the formation of the computation network in the VXU is complete and ready. The SPU specifies the sequence of instructions corresponding to a vector operation by giving the CM address of the first instruction in the sequence, and then activates the sequencer. The last instruction of the

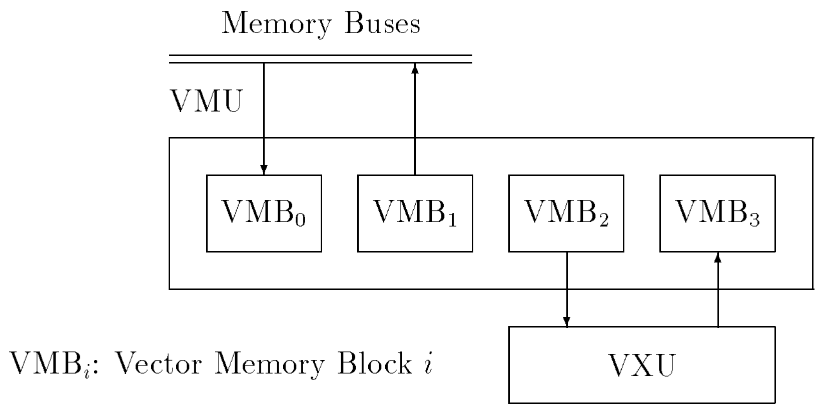


Figure 5.6: VMU Structure.

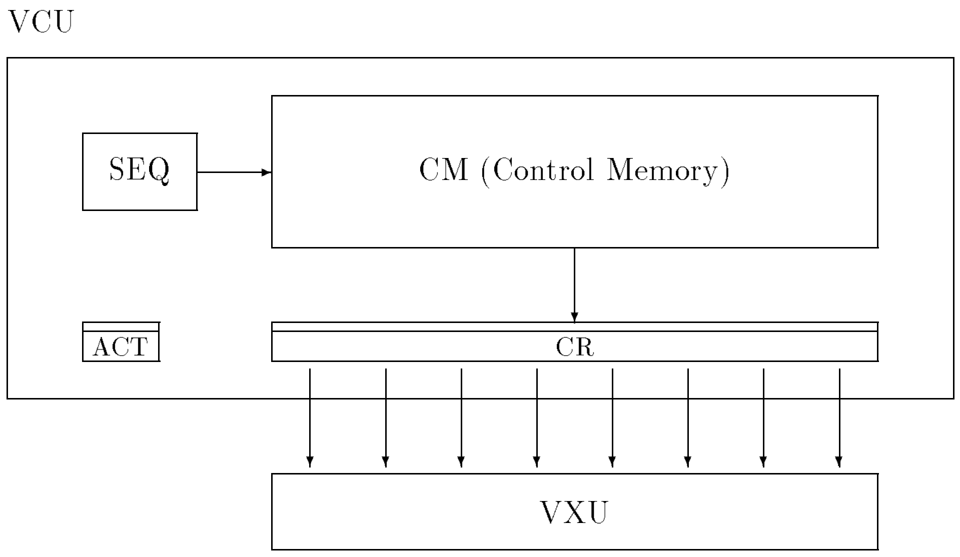


Figure 5.7: VCU Structure.

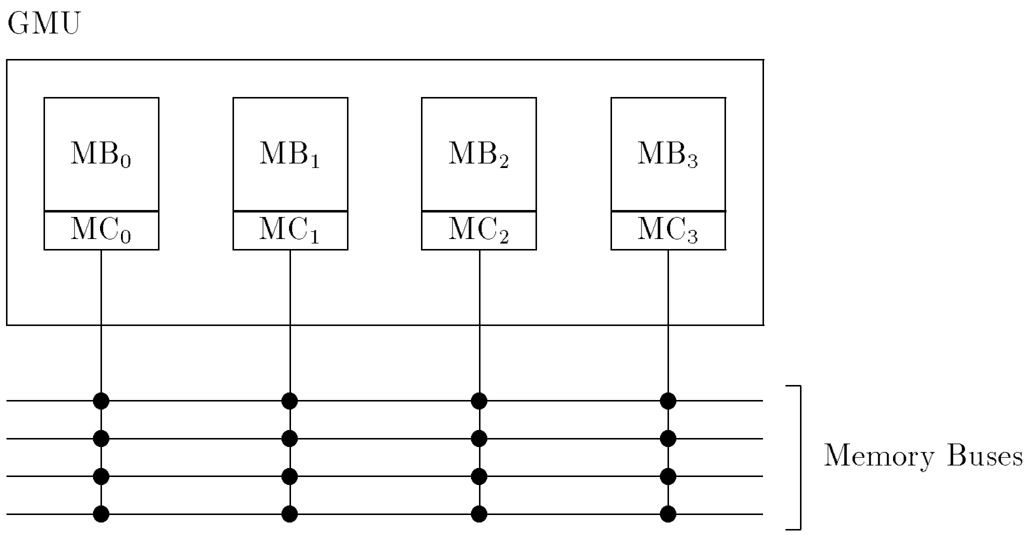


Figure 5.8: GMU Structure.

sequence contains the halt operation, which inactivates the sequencer and signals it to the SPU. The SPU checks to see that the sequence of instructions successfully terminates and then activates the vector operation by turning on the activation flag in the activation control register (ACT). The termination of a vector operation turns off the activation flag in the activation control register and signals it to the SPU by interrupt.

The design spectrum of the VCU is large. One end in the design spectrum is a single instruction per one vector operation. In this case, the instruction contains the control fields for all the components in the computation network, and it stays active until the vector operation terminates. The other end in the design spectrum is some encoding of multiple control functions into one field of the instruction. In this case, there are types of instruction, and the separate registers are required to hold the control data that are active during the vector operation. In order to reduce the setup time for a vector operation, some design inclined towards the first end is desirable.

The VXU is the execution unit of vector operations, where the computation network is organized.

5.4 Global Memory Unit (GMU)

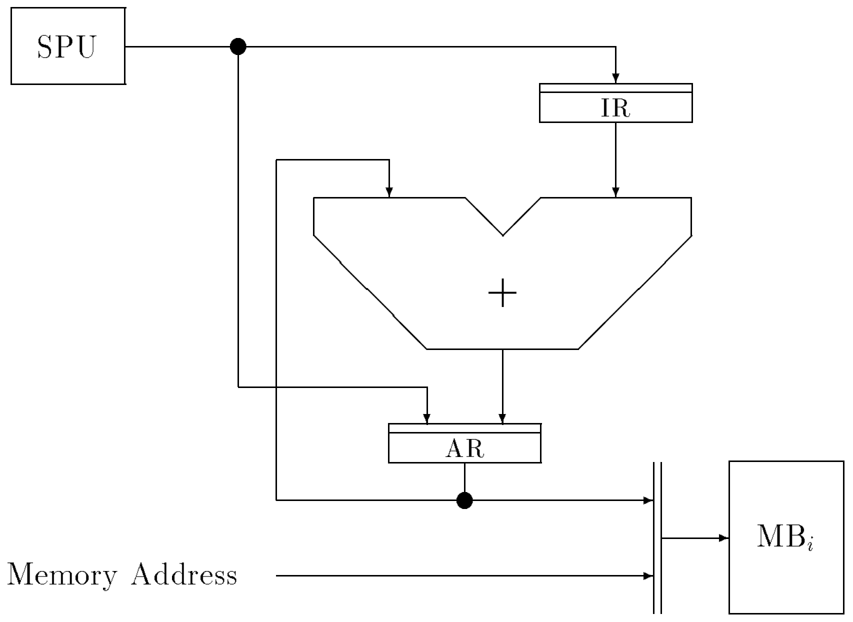


Figure 5.9: GMU Address Generator.

The Global Memory Unit or GMU is a system-wide memory unit and connected to the four memory buses. It is accessed from the other system components. As depicted in Figure 5.8, it consists of four memory banks (MB_0 through MB_3); each bank is equipped with an access controller (MC_i) and can be accessed independently so that four data transfer operations, each associated with a memory bank, can be performed in parallel.

Most of data transfer operations from/to the GMU involve a relatively large quantity of data (vector data) that are stored regularly. A memory address generator is attached to each memory bank to support a DMA-like operation. It is illustrated in Figure 5.9, the register AR holds the current memory address and the register IR holds the address increment to be added to the value in AR for the next memory location. With this address generator, the memory address is automatically updated every time a memory request is received.

5.5 Input/Output Units (IOUs)

The Input/Output Units or IOUs are the input and/or output units. They depend heavily on the I/O requirements. One of the input units is an input unit that takes real-time data at a certain interval. It buffers real-time data and transfers them to a memory bank in the

GMU via one of the memory buses. One of the output units may be an output unit that sends data to a display. It reads data from a memory bank in the GMU through one of the memory buses.

Other possible IOUs include a communication unit, a system diagnostic unit, a disk control unit, a program loading unit, and so forth. The communication unit handles serial-line communications, network-based communications, etc. The system diagnostic unit provides a means for diagnosing the system components. The disk control unit controls one or more disks. The program loading unit is a unit for loading programs from outside the system, when the system is embedded without disks.

5.6 Computation Network

The computation network formed in the VXU consists of functional modules. Basic functional modules are as follows:

- Input Module
- Output Module
- Arithmetic-Logic Module
- Register Module
- Delay Module
- Shift Module
- Multiply Module

The input module reads vector data, element by element, from the VMU (Vector Memory Unit). Figure 5.10 shows the structure of the input module. It contains an address generator that can successively generate linear addresses in the form $b+i*s$, where b is the base address, i the vector index, and s the stride between vector data. The current memory address is held in the register MR, and the address stride in the register SR. It also contains the data counter, which counts the number of vector elements to read. The VMU takes the memory address stored in MR, reads the vector element at that location into the data register DR. The current data count is kept in the register CR. When the data count becomes zero, it will be signalled to the VCU.

The output module stores the output stream of vector data into the VMU. Its structure is shown in Figure 5.11. It contains an address generator and a data counter, both of which are similar to those in the input module. It also contains a stage counter and the activation control. The stage counter initially holds the number of stages through which data flow from an input module to an output module, and counts down every cycle after the input module starts reading data from the VMU. After the number of cycles held in that counter, the output module starts writing the result into the VMU. This is because it takes time

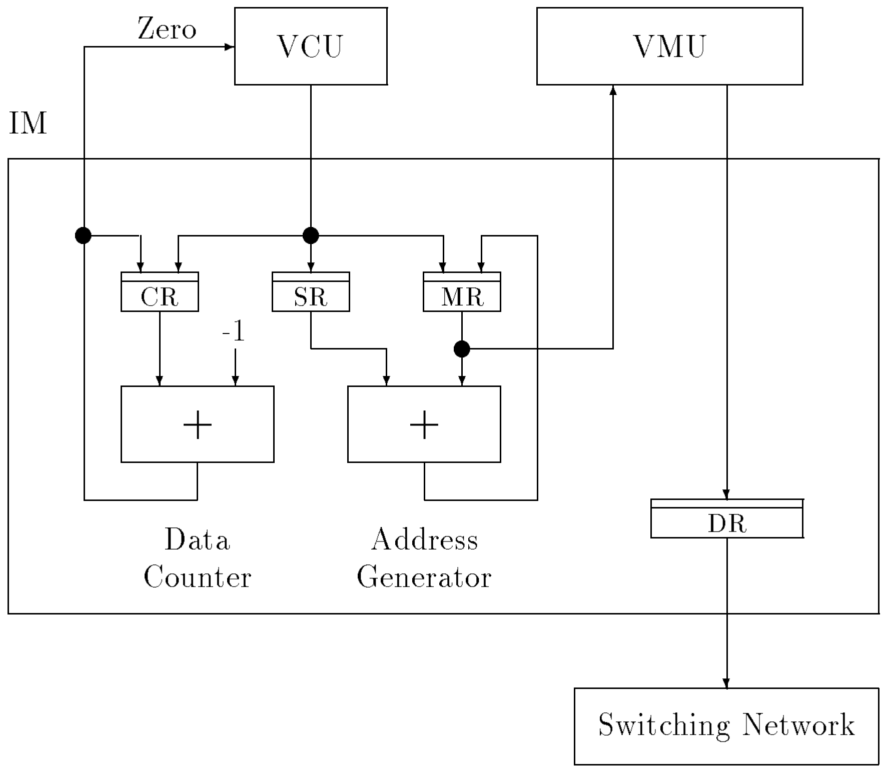


Figure 5.10: Input Module.

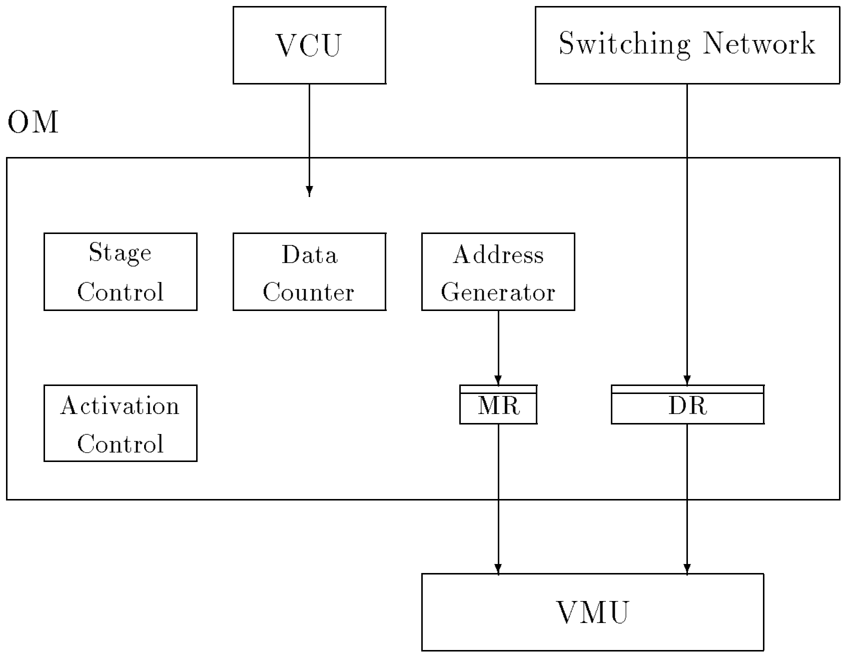
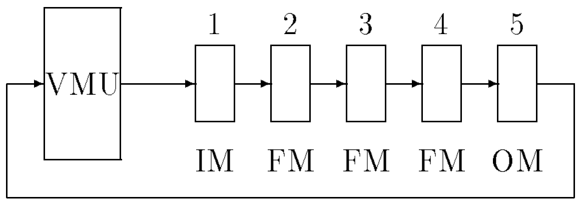
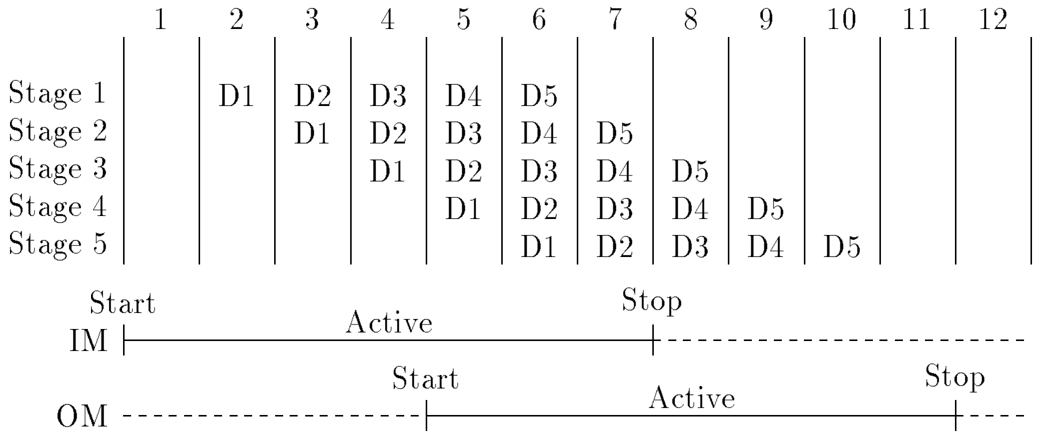


Figure 5.11: Output Module.



(a) 5-Stage Pipeline.



(b) 5-Stage Pipeline Flow.

Figure 5.12: Pipeline Start/Stop Control.

for the first data read by the input module to propagate through the pipeline. When the value in the stage counter becomes zero, implying that the first result has reached to the output module, the output module stores the data in the data register DR into the VMU. The activation control is for the activation/deactivation of the write operations.

The activation/deactivation control of the pipeline is illustrated in Figure 5.12. Figure 5.12(a) shows an example five-stage pipeline. The input module is at the first stage, the output module at the fifth stage, and between the first and last stages are three functional modules. Figure 5.12(b) shows a timing chart, where six data elements (D1 through D6) flow through the five-stage pipeline. The input module becomes active at time 1 and inactive at time 8, whereas the output module becomes active at time 5 and inactive at time 12.

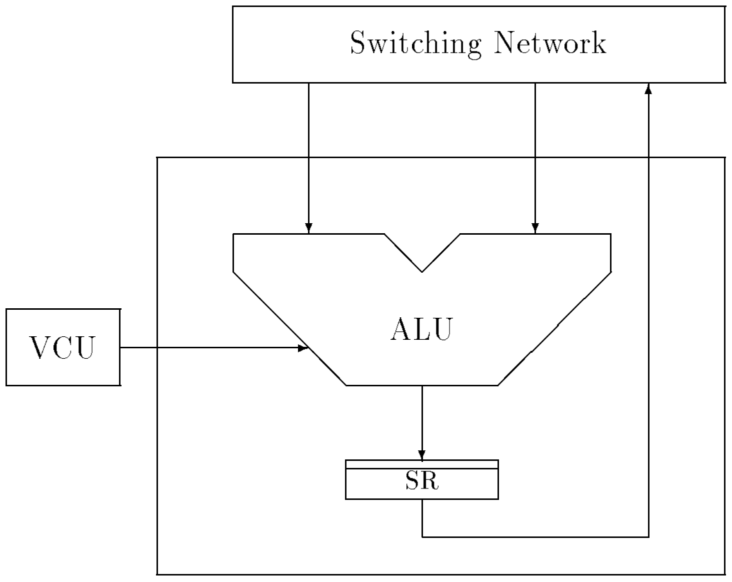


Figure 5.13: ALU Module.

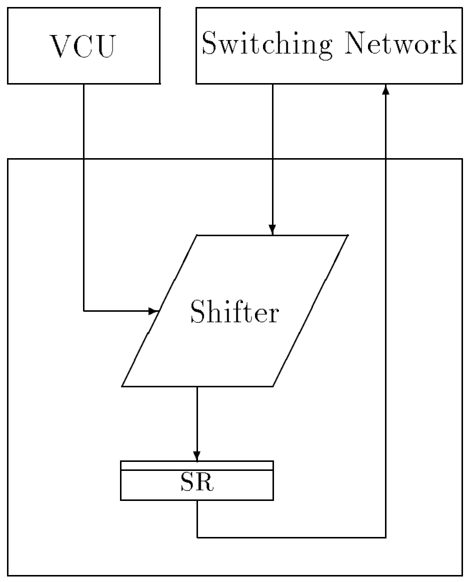


Figure 5.14: Shift Module.

The arithmetic-logic module performs arithmetic and logic operations. Its structure is shown in Figure 5.13. It takes two operands from the switching network and performs the arithmetic or logic operation specified by the VCU. The result of the operation is stored in the staging register SR, which is connected to the switching network.

The shift module performs an arithmetic or logical shift operation. Its structure is illustrated in Figure 5.14. It takes the operand from the switching network and stores the result in the staging register SR, which is connected to the switching network. The shift control (arithmetic/logical, right/left, shift amount) comes from the VCU.

The register module contains a set of registers to provide a constant or to store an intermediate data. Its structure is shown in Figure 5.15. The read/write control and the register address are specified by the VCU. If it is in a read mode, the data stored at the specified register is read into the staging register SR. If it is in a write mode, the data coming from the switching network is stored into the specified register. The registers can also be used as an accumulator in a read/write mode. In one pipeline cycle, the data stored in the register specified by the read address is read into the staging register, and at the same time the data from the switching network is stored into the register specified by the write address.

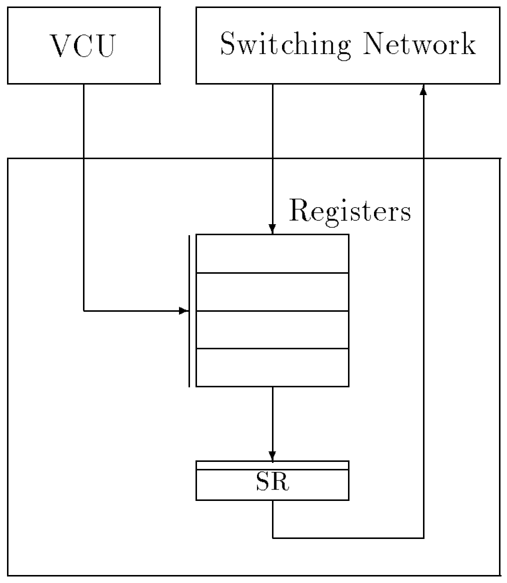


Figure 5.15: Register Module.

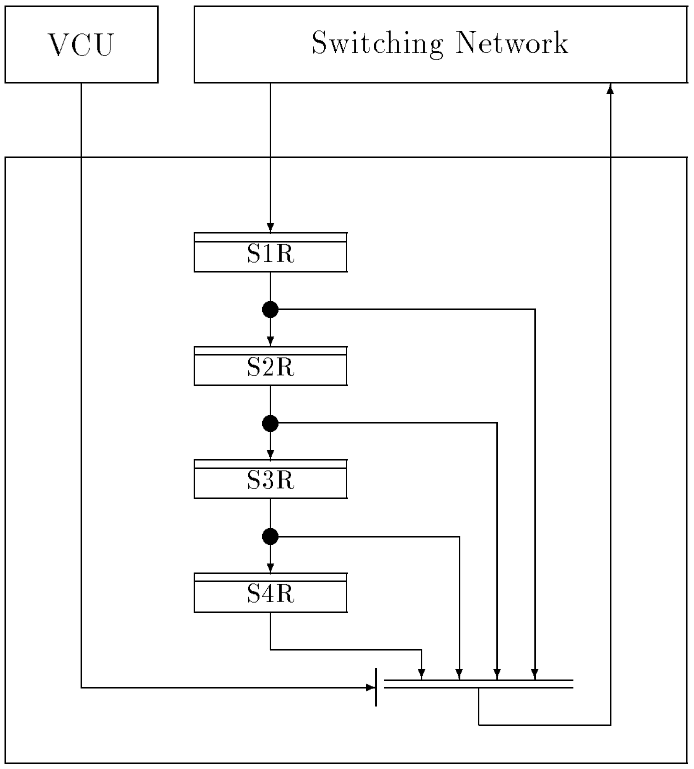


Figure 5.16: Programmable Delay Module.

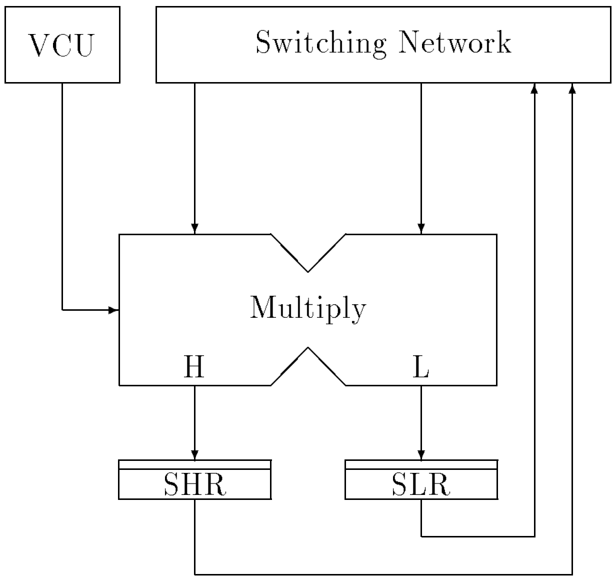


Figure 5.17: Multiply Module.

The delay module is a programmable staging register module. An example delay module is shown in Figure 5.16, where four staging registers are contained and one of them is selected by the VCU.

The multiply module multiplies two data. Its structure is shown in Figure 5.17. Two operands are taken from the switching network, and the higher bits of the product are stored in the register SHR, and the lower bits in the register SLR. Control is provided by the VCU.

Each of the basic functional modules produces results in one pipeline cycle. Functional modules may take more than one pipeline cycle. For example, a floating-point adder and a floating-point multiplier are multi-cycle functional modules. In order to keep the principle of pipeline, they have to perform floating-point operations in pipeline. The design for pipelining of floating-point arithmetic has been well known. Similarly, a divide module and a multiply module for longer data are also multi-cycle modules that can operate in pipeline. See [83], for example, for more detail.

There are many possible interconnections to connect functional modules [168], [1], [194]. One of the simplest interconnections is the bus-based interconnection. The number of data buses depends on how many data transfer operations occur between functional modules at the same time. If the number of data buses is small, all the functional modules can be connected to all the data buses. However, if the number of data buses is large, it is not feasible to connect all the functional modules to all the data buses. If data transfer operations can be divided into groups, then a technique to cluster data buses may reduce the number of connection points. In this case, each group of data transfer operations corresponds to each cluster of data buses. Inter-cluster data buses are also required to carry data from one cluster to another.

One approach to reducing the connection points between functional modules and data buses is a time-sharing bus technique. In this technique a data bus is time-shared between multiple data transfer operations. It requires subdivision of a pipeline cycle. If a pipeline cycle can be divided into, say, four subcycles, then four data transfer operations can be performed in one pipeline cycle. The implementation of this technique usually requires the faster and expensive technology than that for the other components. It also requires a complex timing control design.

The most general approach to interconnecting functional modules is the multi-stage interconnection network, which operates in pipeline. One of its advantages is that it provides flexibility in interconnecting functional modules. One of its disadvantages is that it makes the total length of pipeline longer. Since every data transfer operation between one functional module and the other requires a certain number of pipeline stages to go through, a long pipeline operation may be made much longer. A long pipeline is undesirable from the viewpoint of performance if the vector length is relatively short (See Section 4.2). Therefore, in general, it is too general for the target class of real-time applications.

5.7 DRA Programming

A programming model of a system is a view from the programmer who develops programs for the system. The programming model of the DRA is defined by the SPU, since all the programs written for a DRA system are executed in the SPU, which controls the activities of the other system components. All the system components controlled by the SPU are mapped in the memory-mapped I/O space of the SPU. They are all assigned unique I/O addresses, to which the SPU sends control data and from which the SPU receives status data.

If the SPU is built around one of the RISC microprocessors, the programs to be executed in the SPU are written in the assembly language or high-level languages for the microprocessor. However, these languages do not provide any support for programming for the vector operations in the VPU, since all the functions of the VPU are controlled via I/O commands. It may be possible to extend the assembly language or the high-level languages so that they can directly handle the functions of the VPU at a high level. However, it usually takes a long time to develop such an assembler or a compiler, and in many cases some language constructs force programmers to follow a particular style of programming, which is not necessarily favorable to them in terms of programming efficiency. Because of these reasons, it is more practical and reasonable to develop libraries linked to the object codes produced by the assembler or the high-level language compiler. The libraries are basically sequences of I/O commands, mixed with some arithmetic instructions.

For user-level programming of vector operations in the VPU, the following three functions can be used:

- **StartVF**();
- **CheckVF**();
- **AbortVF**();

where the **StartVF**() function starts the vector function specified by its arguments, the **CheckVF**() function gets the status of the current vector function executed in the VPU, and the **AbortVF**() function aborts the current vector function.

The **StartVF**() function is an asynchronous function; it returns before the operation terminates. The three functions are described in Figure 5.18. As shown in the figure, all the three functions are basically the same in structure. They first finds the library function and sets up the parameters using the template defined by the library function merged with the parameters given as their arguments. Then they issue a trap to the system to transfer control to the VPU handler through the system trap routine. The VPU handler issues I/O commands to the VPU according to the given parameters, and obtains the status at the end of the operations, which is returned to the program. This is illustrated in Figure 5.19.

It is very hard for a programmer to make a program to control the VPU, since the program is basically just a sequence of I/O commands. Programming tools would help the programmer write programs for the VPU. They include a vectorizing tool and a debugging and performance tuning tool.

The vectorizing tool is a software tool that vectorizes a program written in a high-level programming language like Fortran, Pascal, or C. The vectorizing techniques have been studied in [101], [144], [5], [190]. They are based on the dependence analysis [99, Chapter 2, pp.80–186], [143], [100], [191], [52], [119]. These techniques have been incorporated into commercial compilers [10], [199], [164], [174], [127], [40]. Based upon these techniques, the vectorizing tool analyzes a program to find the dependency and transforms it to reduce the dependency.

The debugging and performance tuning tool helps the user debug the programs written for the VPU and enhance their execution performance. Many research efforts have been made around parallel programs running on asynchronous multiprocessor systems. Since no assumption is made about the relative speed of parallel processes and their deterministic properties, debugging logical and timing errors is very difficult. There are two approaches

```
int StartVF( fnum, plist ) /* returns the status */
int fnum; /* vector function number */
int *plist; /* parameter list */
{
    Find the library function specified by fnum;
    Set up the parameters from the plist;
    Issue a trap to the system;
    (The system actually issues I/O commands to the VPU)
    Return the status at the return from the trap;
}


int CheckVF( fnum ) /* returns the status */
int fnum; /* vector function number */
{
    Find the library function specified by fnum;
    Set up the parameters for status check;
    Issue a trap to the system;
    (The system issues I/O commands to the VPU)
    Return the status at the return from the trap;
}


int AbortVF( fnum ) /* returns the status */
int fnum; /* vector function number */
{
    Find the library function specified by fnum;
    Set up the parameters for abort;
    Issue a trap to the system;
    (The system issues I/O commands to the VPU)
    Return the status at the return from the trap;
}
```

Figure 5.18: **StartVF**(), **CheckVF**(), and **AbortVF**().

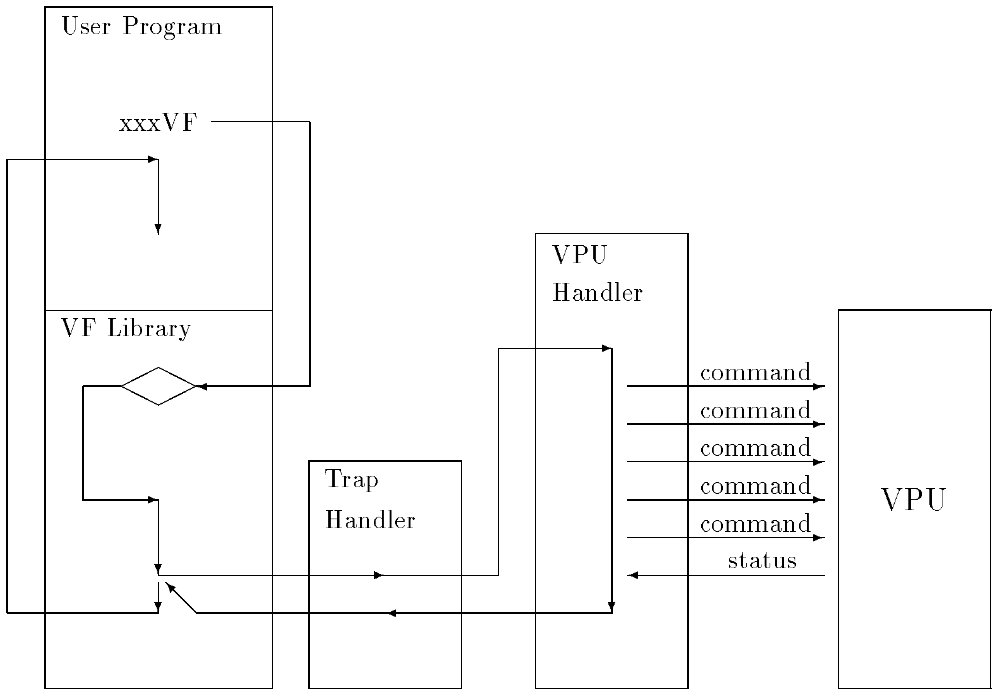


Figure 5.19: Vector Function Execution Flow.

to debugging parallel programs: the selective snapshot approach [18], [13], [33], and the replay approach [32], [38], [112]. There is a unified approach to program debugging and performance tuning based on replay [135] on the BBN Butterfly processor [136].

Unlike general multiprocessor systems, programs running on pipeline systems are easier for debugging and performance tuning, because their behavior is synchronous and deterministic. Although some hardware mechanism for debugging and performance tuning is useful, a simulator for the system should be enough. Since the design philosophy of the DRA includes the avoidance of the statistical behavior, the system simulator can simulate the behavior of the DRA system to an arbitrary accuracy. Therefore, the system simulator is also used as a debugging and performance tuning tool for a DRA system.

5.8 Implementation Considerations

In implementing a DRA system, the major design effort would be required to implement the VPU, where the computation network is formed. The types and numbers of functional modules in the VXU are the most important design parameters to determine. A general approach to their determination is to analyze the set of applications for the DRA system to identify the vector computations, and to determine the set of functional modules that can be used to form the pipelines required for all the vector computations with timing constraints. It is easy to identify the types of the functional modules (arithmetic-operation-level computational resources) required for the vector computations. However, the numbers of the functional module types are not necessarily easy to determine.

Suppose that a set of vector computations is given by $S = \{V_1, V_2, \ldots, V_n\}$, where each vector computation V_i is a pipeline sequence of arithmetic operations given by $< v_i^1, v_i^2, \ldots, v_i^{m_i} >$. The types of the functional modules can be determined from all the different operations found in each sequence of V_i. We number the types of functional modules from 1 to k. Let F_i be a list of the form $(f_i^1, f_i^2, \ldots, f_i^k)$, where f_i^j is the number of functional module type j required for the vector computation V_i. Let G be a list $(g_1, g_2, \ldots, g_k)$, where $g_j = \mathbf{max}_{i=1}^{n}\{f_i^j\}$. Then the list G shows the number of each functional module type required for the set of vector computations given by S.

The pipeline sequence of arithmetic operations given by the form $< v_1, v_2, \ldots, v_m >$ is a single-input single-output sequence. However, a vector computation may be a multiple-input multiple-output sequence in a general case. Then it is represented by a graph (a pipeline graph), as described in the previous chapter. But the number of each functional module type can be computed in the same way as before. Therefore, for a given set of vector computations, each represented by a pipeline graph, we can identify all the types of functional modules and compute the number of each type with a relatively simple method.

If a single vector computation requires a large set of computational resources, it needs to be divided into smaller vector computations so that the total number of functional modules can be reduced. It is not obvious to divide a vector computation to optimize the number of functional modules; it is an optimization problem. Some research work is required to address this problem, which is beyond the scope of our work. If a large vector computation is critical for the application, it is sometimes very difficult to divide it because of timing requirements. Then it has to be implemented anyway, no matter how many functional modules are required and no matter how inefficient they are in terms of usage.

The interconnections of the functional modules are also an important design problem. A cross-bar switch can provide all the possible interconnections for a given set of functional modules, but it is very expensive. The analysis of the pipelines formed for the vector computations may reduce the complexity of the interconnections. When the number of possible interconnections are relatively small, then multiple data buses can be used to connect functional modules. Data buses are a relatively cheap solution. If the interconnections can be divided into some groups, then clusters of data buses may be useful. In general, some form of switching network is chosen depending on the interconnection requirements. There is no well-known method to give a switching network for a given set of interconnections. Exploring this method also requires some research effort.

There would be variations in the system-level structure. The basic system structure is built around four memory banks and four data buses so that four data transfer operations can be performed in parallel. Some applications may require an extra data transfer operation, for example, for communications with the other computing systems. Other applications may need more memories and data buses for extra input and/or output. Since the basic principle in the design of a DRA system is to keep the system-level pipeline in order to flow a large amount of data without any blockage, all the variations must be implemented to keep this principle. As long as the principle is kept in design, system-level pipeline processing is guaranteed.

When a real-time application contains a large set of vector computations, it is very difficult to design a system manually. A possible useful tool might be a design tool to help a system designer design the computation network for a given set of vector computations by computing the types and numbers of functional modules and giving their interconnections. Some research work is required to explore such a design tool.

Since all the vector computations are performed with pipelines formed in the computation network, their performance is predicted as described in Section 4.2 (Performance Models). If the application is composed of vector computations and a small amount of glue scalar operations, its performance is predicted to be high. However, if the application contains many scalar operations, then its performance is not expected to be high in general, because of Amdahl's law. If scalar operations in the SPU can be overlapped with vector operations in the VPU, the performance will not be degraded much. Conversely, as many scalar operations as possible need to be performed in parallel with vector operations in order to avoid the performance degradation. The performance models described in Section 4.2 give a general guideline for the performance design of the application mixed with scalar and vector computations.

The execution model described in Section 4.3 serves as a theoretical model for multiple pipelines, on which the proposed new architecture is based, and has shown that a pipeline can be configured for a given vector operation. It also has shown the techniques for vector reductions and linear recurrences with some restrictions. The described techniques would be used to build a tool to automate the pipeline design process. For a given computation network, the tool would map a vector operation onto a pipeline.

Chapter 6

DRA Example Implementation

This chapter presents the MSP (Mitsubishi Signal Processor), an example implementation of the *Dynamically Reconfigurable Architecture* or *DRA*, which was designed as an experimental real-time signal processor primarily for high-speed real-time radar signal processing [134], [137], [138], [139].

6.1 MSP Structure

The MSP is an instance of the DRA system. Its structure is illustrated in Figure 6.1. The GMU (Global Memory Unit) is the system-wide memory unit that stores real-time input data, intermediate data, and result data. The CPU (Control Processor Unit) is the scalar processor that controls the other system components. It contains the PMU (Program Memory Unit) and PCU (Program Control Unit). The PMU stores programs to execute, and the PCU executes the programs. The APU (Array Processor Unit) is the vector processor that performs vector operations. It contains the DMU (Data Memory Unit) and the DPU (Data Processor Unit). The DMU stores vector data and the DPU performs vector operations with the dynamically reconfigured computation network under the control of the CPU. The XIU (External Interface Unit) provides the interface to the outside world. The IOUs (Input/Output Units) are the input/output units. The M-Bundle consists of four data buses and connects the GMU, the CPU, the APU, and the XIU, which are also connected via the C-Bundle consisting of two control buses. The IO-Bundle consists of two data buses and connects the XIU and the IOUs.

The main data handled by the MSP are 24-bit integers and 24-bit complex data with 12-bit integer real and imaginary parts. The MSP does not handle floating-point data. The data buses in the M-Bundle and IO-bundle are all 24 bits wide. The internal structures of the APU and the CPU are also 24 bits wide.

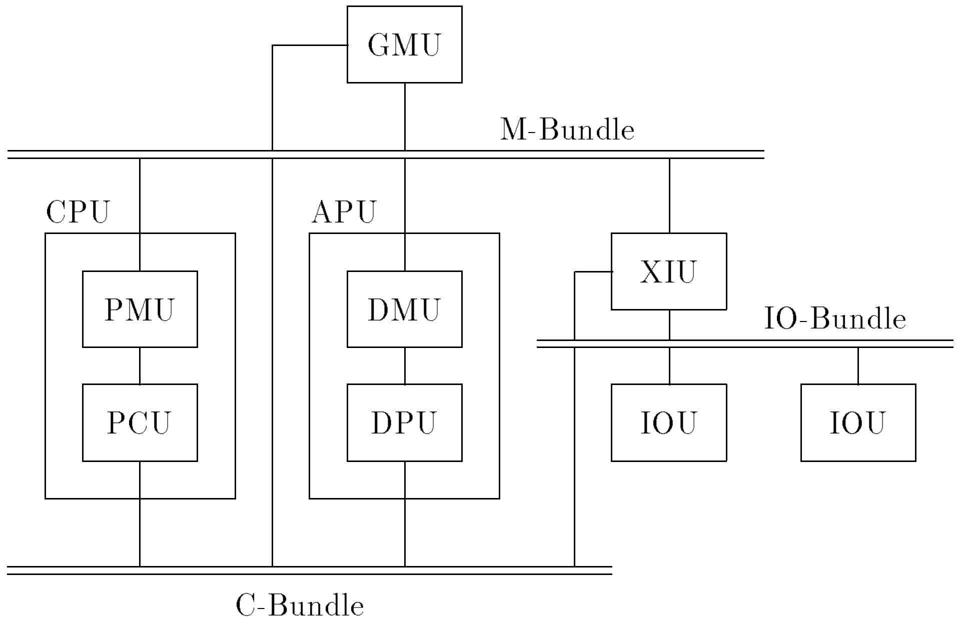


Figure 6.1: MSP Structure.

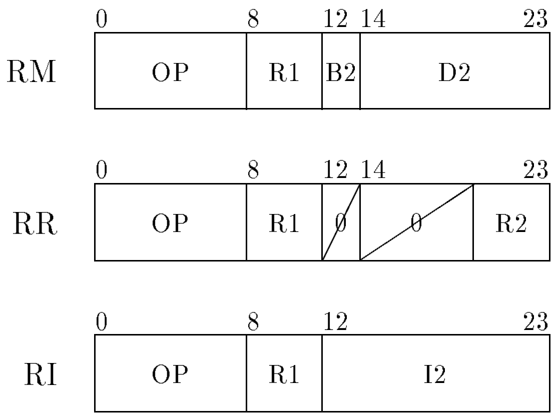


Figure 6.2: MSP Instruction Formats.

6.2 CPU Structure

The CPU is an implementation of the SPU (Scalar Processor Unit) of the DRA. The LMEM is a logical memory local to the CPU and stores programs and data. It is implemented in the PMU. It is accessed on the basis of 24-bit data.

There are four types of registers:

- 16 general registers (GR0-F);
- four base registers (BR0-3);
- four control registers (CR0-3); and
- the program status register (PSR).

The general registers GR0-F are used to hold intermediate data and two of them (GR0-1) are also used to as index registers. The base registers BR0-3 are used to store base addresses for memory accesses. The base register 0 (BR0) always contains the base address for instruction fetch. The control registers CR0-3 hold system control data including interrupt control bits and address search data. The program status register PSR is 48 bits long and holds the instruction address, operation status, interrupt masks, interrupt levels, and so forth.

The machine instructions executed in the CPU are all 24 bits long and categorized into three format types:

- RM (Register-Memory) format;
- RR (Register-Register) format; and
- RI (Register-Immediate) format.

These three formats are illustrated in Figure 6.2. The instruction fields are as follows:

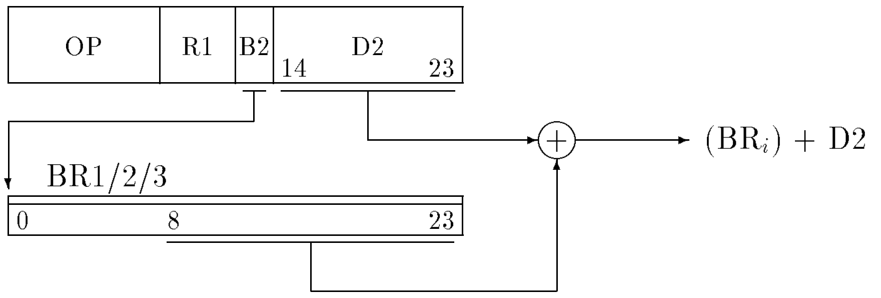
- OP (bits 0-7): operation code;
- R1 (bits 8-11): general register for the first operand;
- B2 (bits 12-13): base register for the second operand;
- D2 (bits 14-23): address displacement for the second operand;
- R2 (bits 20-23): general register for the second operand;
- I2 (bits 12-23): immediate data used as the second operand.

The first operand is stored in the general register specified by the R1 field, where the result of the operation is also written. The second operand is stored at the LMEM location pointed by the base register specified by the B2 field and the displacement specified by the D2 field in the RM format, stored in the general register specified by the R2 field in the RR format, or is the value of the I2 field in the RI format.

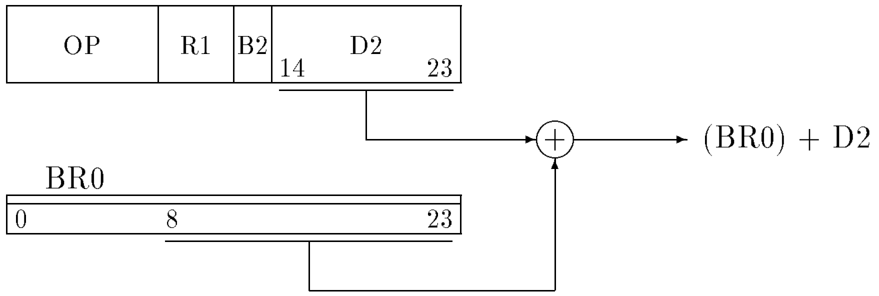
The memory address generations for the second operand in the RM format, the branch target address in the RM format, and the next instruction address are shown in Figure 6.3. The generated memory addresses are all 16-bit addresses of the LMEM. The second operand address in the RM instruction is formed by adding the displacement in the D2 field to the address held in the base register specified by the B2 field. Three base registers BR1-3 are available for the second operand. The branch target address in the RM instruction is formed by adding the displacement in the D2 field to the address stored in the base register 0 (BR0). The instruction address used to fetch the next instruction is formed by adding the value stored in the base register 0 (BR0) to the value in the instruction address field of the program status register (PSR). The base register 0 (BR0) always holds the current program base address. The value kept in the PSR is relative to the base address in the BR0.

The LMEM is a memory local to the CPU. Data and programs can be transferred back and forth between the global memory (GMEM) in the GMU and the LMEM by the LDL (Load Local Memory) and STG (Store Global Memory) instructions. The LDL instruction loads data/programs from the GMEM to the LMEM; the STG instruction stores data from the LMEM to the GMEM. Both of the instructions move the specified amount of data/programs in the manner similar to the DMA operation. Both instructions are of the RM type, and their second operand specifies the parameter list of four words to control the data transfer operation. The parameter list includes

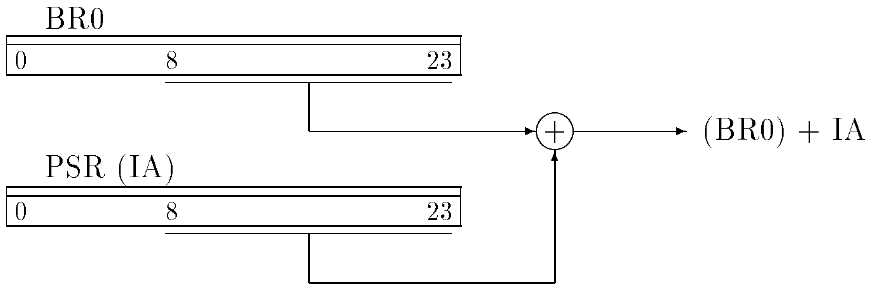
- GAI: GMEM Address Increments;
- GSA: GMEM Start Address;



(a) Operand Address Generation.



(b) Branch Address Generation.



(c) Instruction Address Generation.

Figure 6.3: MSP Address Generations.

- LSA: LMEM Start Address; and
- WC: Word Count.

This parameter list enables the LDL and STG instructions to access the GMEM area represented by GSA + $i \cdot$ GAI ($i = 0, 1, 2, \ldots,$ WC $- 1$). The data transfer operation initiated by these instructions is performed in parallel with the execution of the subsequent instructions and terminates with an interrupt. The TST (Test) instruction can check the status of the data transfer operation.

The structure of the CPU is shown in Figure 6.4. The CPU is a microprogram-controlled processor. The instruction address register IAR holds the next instruction address that is sent to the LMEM. The instruction address held in the IAR is pushed into the IA stack when the subroutine call instruction is executed. The instruction address at the top of the stack is loaded into the IAR when the subroutine return instruction is executed. If the subroutine nesting is too deep for the IA stack to hold all the return addresses, the subsequent return addresses are stored in the LMEM. When the subroutine call is executed and the IA stack overflow occurs, the overflow flag is turned on, and the overflow trap is raised to transfer control to the IA stack overflow routine, which stores the return address in the LMEM. When the subroutine return instruction is executed while the overflow flag is on, the trap is raised to transfer control to the overflow routine, which loads the return address from the LMEM. If there is no more return addresses stored in the LMEM, then the overflow flag is turned off.

The instruction read from the LMEM is loaded into the instruction register IR, where the instruction is decoded. The instruction mapper IMAP generates the microcode entry address for the instruction execution. The microinstructions are stored in the micro memory and loaded into the microinstruction register MIR.

The execution section of the CPU contains a 24-bit ALU, a 24-bit shifter, and a 24-bit data memory containing general registers. There are four data registers AR, BR, CR, and DR. AR and BR are mainly used to hold data for some arithmetic. CR and DR are mainly used to hold the control data (UCC and UCD) to the other system components. The unit control command (UCC) gives the address of the unit to control; the unit control data (UCD) gives the data to the unit. Both of them are sent through the C-Bundle. The control of the APU is provided by the UCC and UCD.

6.3 APU Structure

The APU is an implementation of the Vector Processor Unit or VPU in the DRA. Figure 6.5 shows its structure. It consists of the DMU (Data Memory Unit), the DPU (Data Processor Unit), and the DCU (Data Control Unit). The DMU, DPU, and DCU correspond to the

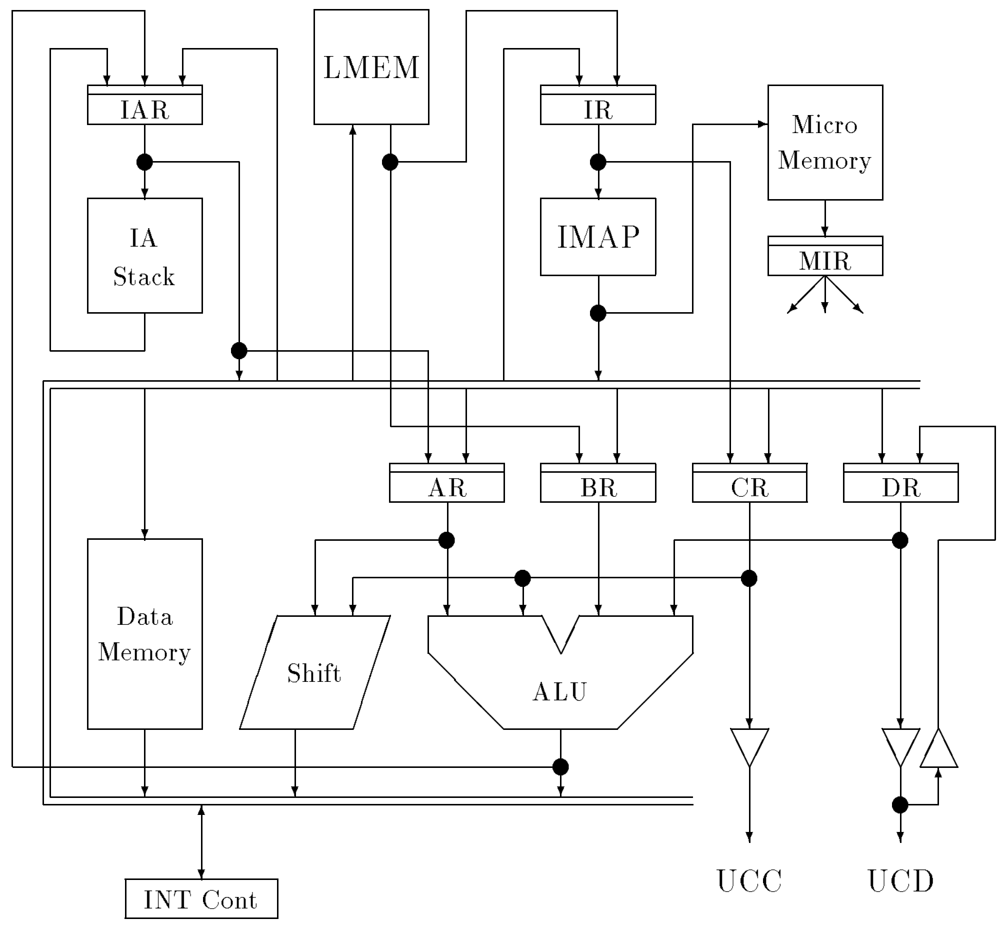


Figure 6.4: CPU Structure.

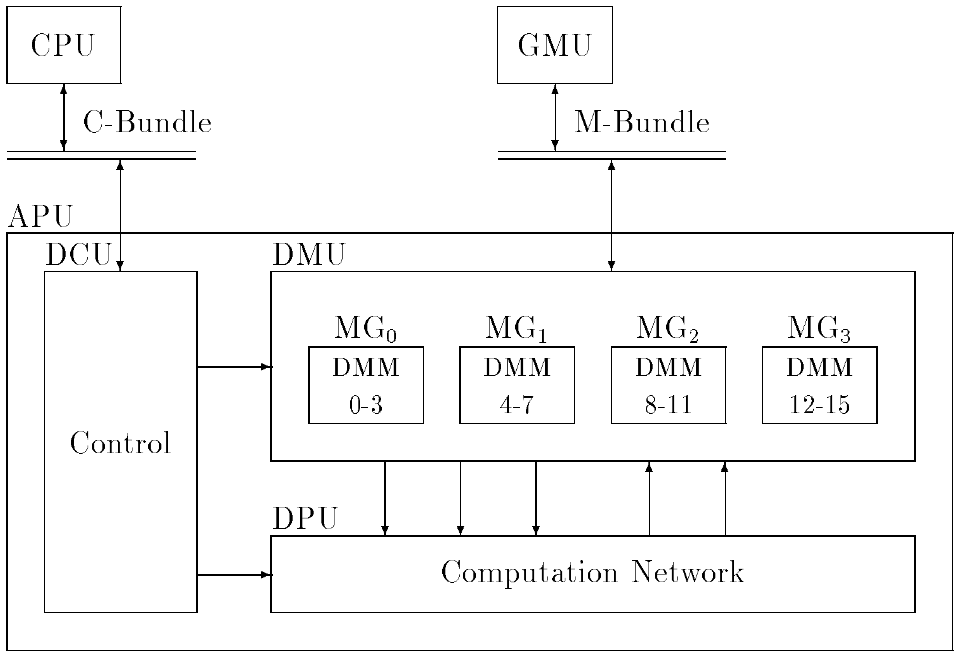


Figure 6.5: APU Structure.

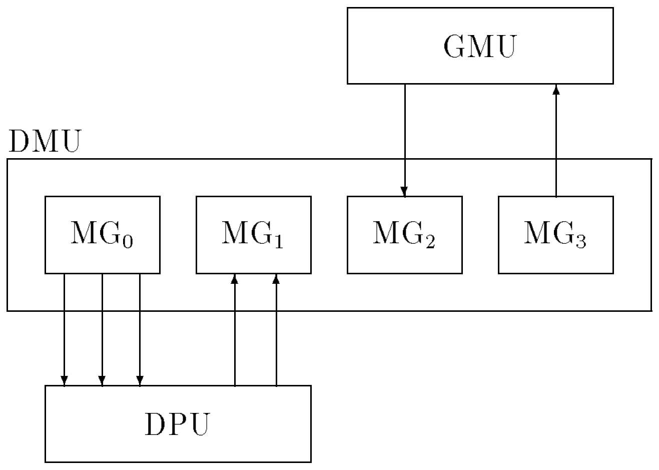


Figure 6.6: Parallel Data Transfer Operations with DMU.

VMU (Vector Memory Unit), the VPU (Vector Processor Unit), and the VCU (Vector Control Unit) of the DRA, respectively. The functions of the DCU are more primitive than those of the VCU, since the CPU has more direct control over the vector computations performed in the DPU.

As discussed in Chapter 5, the capability of parallel data transfer operations is crucial to the system-level pipeline operations. The DMU supports the parallel vector data transfer operations with four groups of data memories MG_0 through MG_3, as shown in Figure 6.6. Each group consists of four data memories (DMMs); the total number of data memories is 16. The four groups of data memories operate independently. In Figure 6.6, MG_0 supplies vector data to the DPU; MG_1 stores the vector data coming out of the DPU; MG_2 stores the input vector data from the GMU; and MG_3 holds the results of vector operations, which are stored into the GMU.

The computation network is formed in the DPU and is controlled by the CPU. The physical structure of the DPU is shown in Figure 6.7. The DPU contains the following types of functional modules:

- IN: Input
- OUT: Output
- PCI: Program Control Interface
- ALU: Arithmetic and Logic Unit

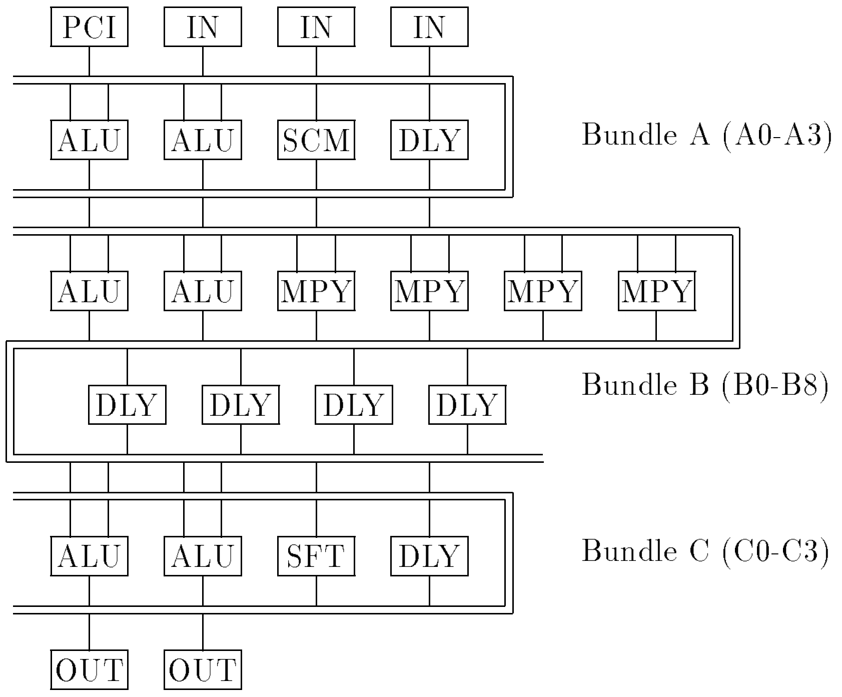


Figure 6.7: DPU Structure.

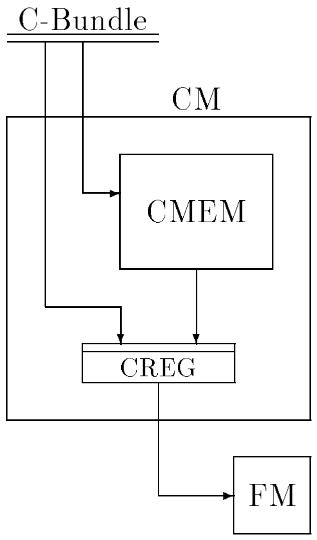


Figure 6.8: Control Module Structure.

- SFT: Shifter
- MPY: Multiplier
- SCM: Scratch Memory
- DLY: Delay

The IN module reads vector data from the DMU. The OUT module writes vector data into the DMU. The PCI module provides interface to the CPU. The ALU module performs one 24-bit integer arithmetic/logic operation or two 12-bit integer arithmetic/logic operations. The SFT module shifts a 24-bit data left or right. The MPY module multiplies two 12-bit integers and produces a 24-bit integer as a product. The SCM module is a scratch memory that holds scalar data. The DLY module is a delay module that contains a staging register.

The DPU consists of the three subnetworks of functional modules. The subnetwork A contains one PCI, three DINs, two ALUs, one SCM, and one DLY, which are all connected via the Bundle A (Buses A0-A3). The subnetwork B contains two ALUs, four MPYs, and four DLYs, which are all connected via the Bundle B (Buses B0-B8). The subnetwork C contains two ALUs, one SFT, one DLY, and two DOUTs, which are all connected via the Bundle C (Buses C0-C3). Each of the data buses consists of two 12-bit data buses. It is usually used a single 24-bit bus; it can be separated into two 12-bit buses when 12-bit real and 12-bit imaginary parts of 24-bit complex data are handled separately.

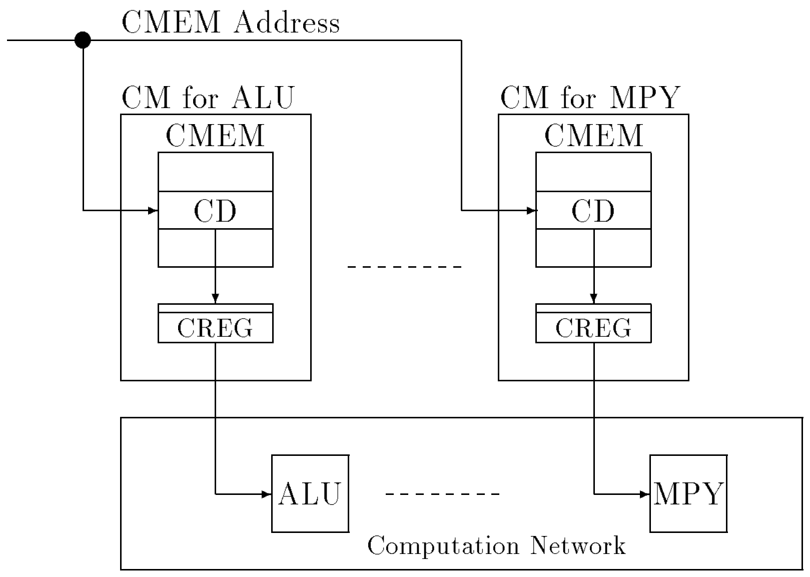


Figure 6.9: Computation Network Configuration Control.

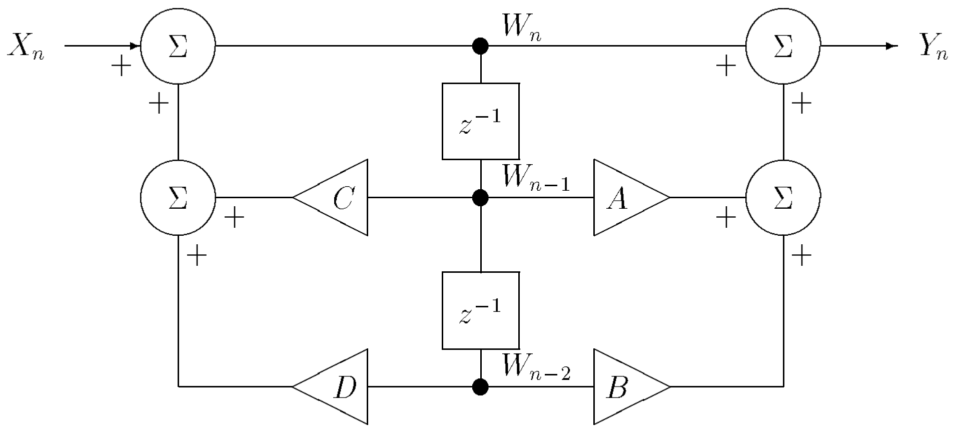


Figure 6.10: 2nd-Order IIR Filter.

In the MSP, a control module is associated with each functional module. Its structure is shown in Figure 6.8. It contains the CMEM (control memory) and the CREG (control register). The CREG holds the control data (CD) for the corresponding functional module during a vector operation. The CMEM stores a set of control data. The address of a CD to load into the CREG is specified by the CPU through the C-Bundle. In switching from one vector operation to the other, a CD is loaded from the CMEM into the CREG. The CPU can directly load a CD into the CREG via the C-Bundle.

The configuration of the computation network is controlled collectively by the control modules, as illustrated in Figure 6.9. In order to reduce the setup time for a vector computation, a single CMEM address, associated with a vector computation, is provided to all the CMEMs by the CPU. All the CMEMs read a CD at the location specified by the address and load it into the CREGs. It takes only a single step to load CDs required to set up the computation network. Thus, all the CDs at a CMEM location collectively form a single long instruction to control the computation network, as discussed in the previous chapter.

6.4 Filtering

We show the 2nd-order IIR filter operation as an example vector computation to demonstrate how the computation network works.

The 2nd-order IIR filter can be defined by

$$H(z) = \frac{z^2 + A \cdot z + B}{z^2 - C \cdot z - D} . \tag{6.1}$$

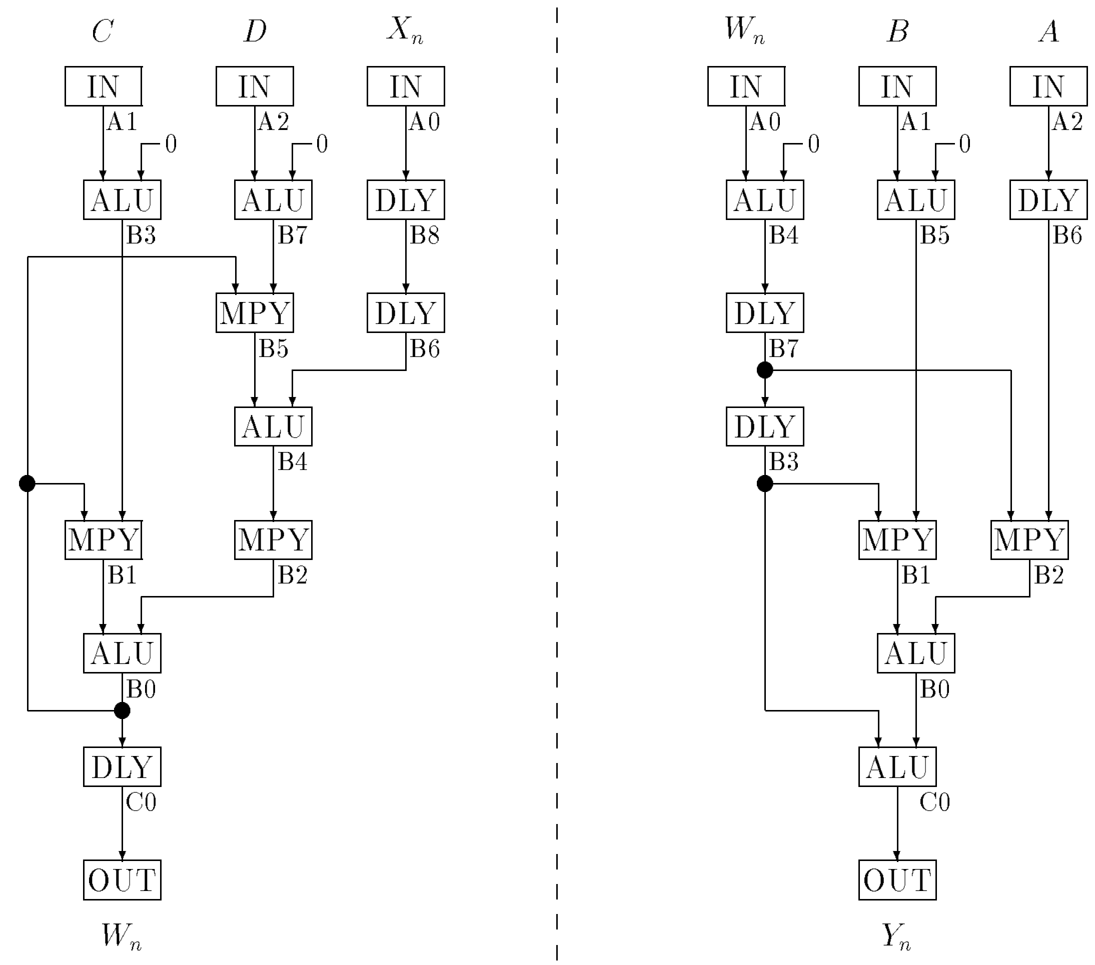
This is illustrated in Figure 6.10. Its computation is given by

$$\begin{aligned} W_n &= X_n + C \cdot W_{n-1} + D \cdot W_{n-2} , & (6.2) \\ Y_n &= W_n + A \cdot W_{n-1} + B \cdot W_{n-2} . & (6.3) \end{aligned}$$

This computation requires two additions and two multiplications for W_n and another two additions and two multiplications for Y_n. Since the computation network of the MSP includes six ALUs and four multipliers, the computational resources are enough to compute W_n and Y_n at one time. However, they can not be computed at one time because there are fewer data buses in the MSP computation network than required. Therefore, W_n and Y_n must be computed separately. Figure 6.11 shows two network configurations: the left network for W_n and the right for Y_n. It also shows the data bus assignments. All the values of W_n are first computed with the left network in Figure 6.11 and stored in memory. Then, the values of Y_n are computed with the stored values of W_n through the right network in Figure 6.11.

Since the equation of W_n is a form of linear recurrence, the computation network contains a feedback loop. It takes two clock cycles to produce each value of W_n in the network; each value of W_n is produced at every other clock cycle. It is due to the fact that add and multiply take two pipeline stages. However, once all the values for W_n are computed and stored in memory, the values of Y_n are computed at the full speed, that is, each value is produced at every clock cycle.

In these networks, all the constants (A, B, C, and D) are read from memory through the IN modules. The address generator of each IN module generates the same address pointing to the location where the constant is stored during the computation. The input data X_n are stored as a vector in memory and read by the IN module with the address generator incrementing the address at each clock cycle. The values of W_n, produced by the left network in Figure 6.11, are stored as a vector in memory and read by the IN module in the computation of Y_n. Since the address generator in the OUT module for W_n updates the memory address by incrementing it by 1 at every clock cycle, the actual values of W_n are stored in alternating positions in the output vector. They are accessed by the IN module with the address generator incrementing the address by 2 so that each value of W_n can be read at every clock cycle in the computation of Y_n.



(a) Network for W_n

(b) Network for Y_n

Figure 6.11: Networks for 2nd-Order IIR Filter.

6.5 FFT Processing

The analysis of the operations in the real-time signal processing applications revealed that the significant amount of time would be spent on FFT computations. In order to accelerate the speed of FFT computations, the MSP employed a special computation scheme based on the dynamically reconfigurable computation network.

Consider a radix-2 FFT algorithm for n data. A radix-2 FFT algorithm has many variations as described in [176]. They differ in computational geometry. The version of the algorithm used in the MSP is shown in Figure 6.12. It is a DIT (Decimation in Time) algorithm that requires a simple data addressing scheme without the bit reverse operation. As shown in Figure 6.12, FFT is computed through "passes." For 2^n data, FFT is computed through $\log n$ passes, each pass containing $n/2$ butterflies. The total number of butterflies amount to $(n/2) \cdot \log n$.

A FFT butterfly (DIT) is defined by

$$\begin{aligned} P' &= P + Q \cdot W, & (6.4) \\ Q' &= P - Q \cdot W, & (6.5) \end{aligned}$$

where P and Q are input data, W is a twiddle factor, and P' and Q' are output data (all the data are complex).

Figure 6.13 shows the network for the radix-2 butterfly computation for FFT. It consists of three INs, two OUTs, six ALUs, four MPYs, and three DLYs. Each of two ALUs following the INs is used as a delay. Each of the two ALUs followed by the OUTs is used as two 12-bit ALUs. In this figure the assignment of data buses is also shown.

Consider the j-th butterfly in pass i, where $0 \le i \le n-1$ and $0 \le j \le 2^{n-1}-1$. Let p, q, p', and q' be the addresses of P, Q, P', and Q', respectively. Then, p, q, p', and q' are given by

$$\begin{aligned} p &= \mathbf{div}(j, 2^{n-i-1}) \cdot 2^{n-i} + \mathbf{mod}(j, 2^{n-i-1}), & (6.6) \\ q &= p + 2^i, & (6.7) \\ p' &= j, & (6.8) \\ q' &= p' + 2^{n-1}, & (6.9) \end{aligned}$$

where for non-negative integers x, y, q, and r, the functions **div** and **mod** are defined as follows:

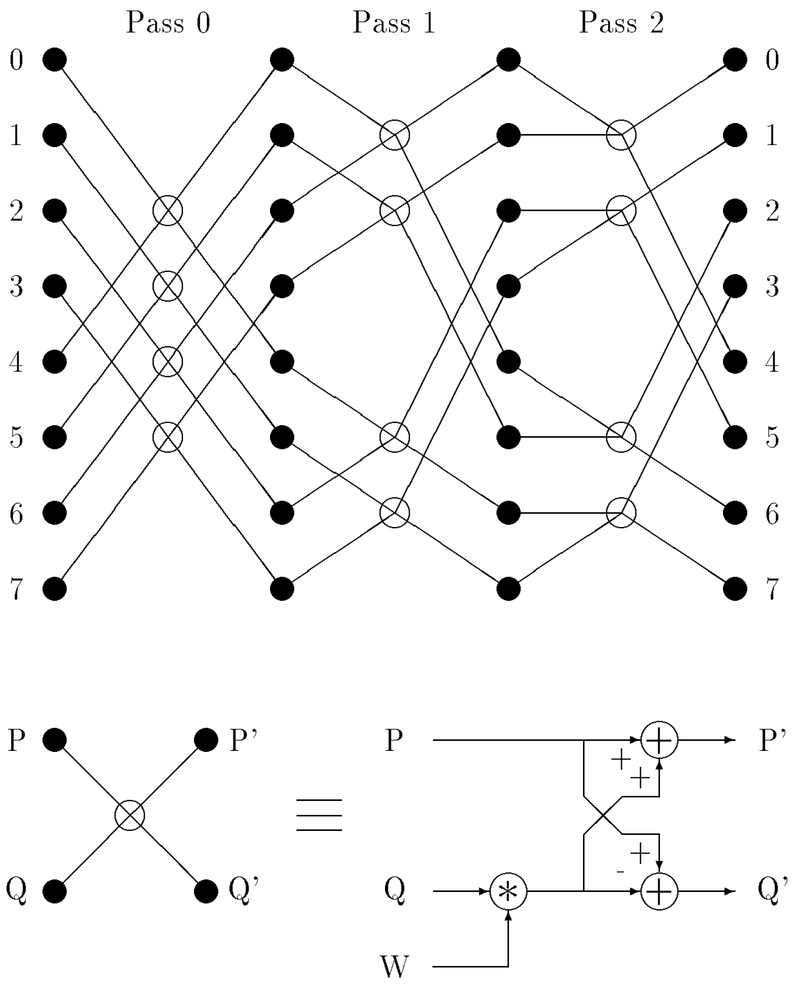


Figure 6.12: FFT Geometry.

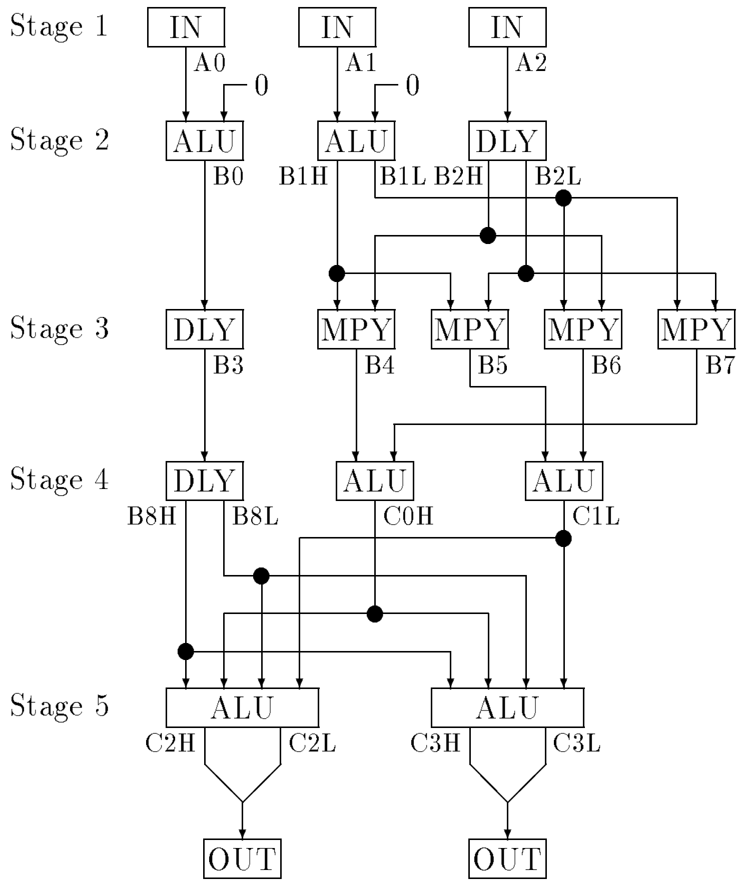


Figure 6.13: FFT Network.

$$\begin{aligned} x &= y \cdot q + r, \quad (0 \le r \le y-1) && (6.10)\\ \mathbf{div}(x,y) &= q, && (6.11)\\ \mathbf{mod}(x,y) &= r. && (6.12) \end{aligned}$$

Each butterfly can be computed with the network configuration shown in Figure 6.13. The butterfly network forms a five-stage pipeline: data read and staging at the first stage, multiplications at the second stage, ALU operations at the third stage, the second ALU operations at the fourth stage, and data write at the fifth stage.

Let $C(n)$ be the number of pipeline cycles required to compute the 2^n-point FFT. Note that each butterfly can be computed in one pipeline cycle with the network configuration shown in Figure 6.13. In general, $C(n)$ is given by

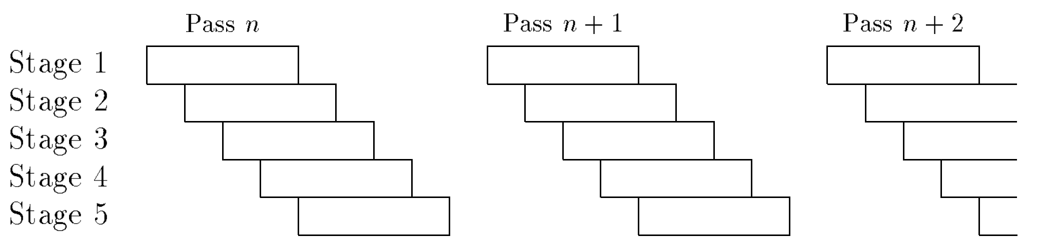
$$C(n) = a + n \cdot (2^{n-1} + b + c), \qquad (6.13)$$

where a is the overhead at the beginning and end of the FFT computation, b is the overhead due to memory contentions, and c is the overhead in transition from one pass to the next.

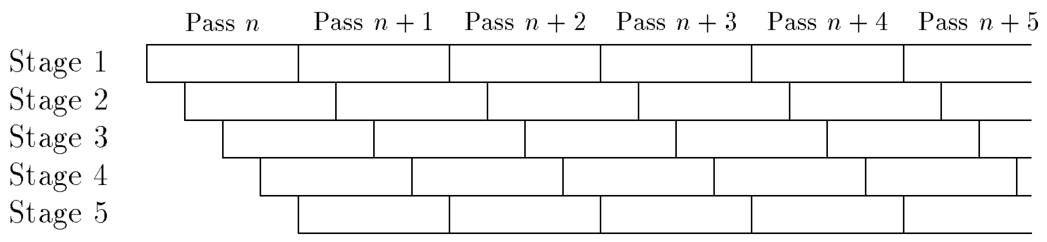
The overhead a is unavoidable as long as the pipeline is used to compute the FFT. The memory contention overhead b is not easily eliminated. Consider a data access scheme in which separate data memories are used for data input and output, and data memories are switched in transition from one pass to the next. The memory interleaving technique with multiple memory banks cannot eliminate memory contentions for the data accesses of p, q, p', and q'. The pass transition overhead c is not easily eliminated, either. Because of the pipeline staging, the pass transition does not take place until all the butterfly results of a pass go through all the pipeline stages and are stored in memory. The MSP employed a new memory access scheme that reduces both b and c to zero. Figure 6.14(a) shows a timing chart for a conventional way of FFT processing with the memory contention and pass transition overheads, whereas Figure 6.14(b) shows a timing chart for the MSP scheme for FFT processing.

In the new scheme, a prime memory system is introduced in order to reduce the memory contention overhead b to zero, and a special data buffering scheme is used to reduce the pass transition overhead c to zero.

The prime memory system was proposed by D. H. Lawrie and C. R. Vora in [111]. The basic idea behind the prime memory system is that vector data stored in the memory system with the prime number of memory banks can be accessed by multiple processors without memory contentions. The MSP uses the mod 3 prime memory scheme, where 2^n vector data are stored in 3 memory modules, as illustrated in Figure 6.15. Since the VMU consists of four groups of four data memories, three data memories in a group can be used for this



(a) Conventional FFT Pipeline Flow.



(b) New FFT Pipeline Flow.

Figure 6.14: FFT Pipeline Flow.

	DMM_0	DMM_1	DMM_2
2	6	7	8
1	3	4	5
0	0	1	2

Figure 6.15: FFT Prime Memory.

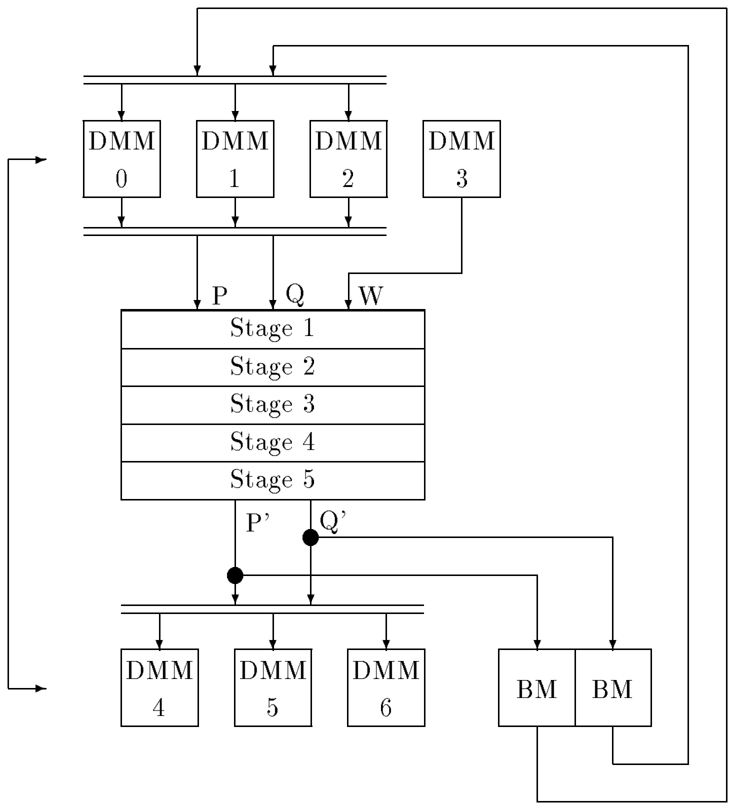


Figure 6.16: FFT Pipeline Configuration.

data arrangement. In one pass of the FFT computation, P and Q are stored in three data memories in one group and P' and Q' are stored in three data memories in the other group. The addresses of P and Q are given by p and q defined by Equations 6.6 and 6.7, respectively. Since the address difference between p and q is 2^i, simultaneous read accesses to P and Q do not cause any memory contentions. Similarly, simultaneous write accesses to P' and Q' do not cause any memory contentions, either, since the address difference between p' and q' is 2^{n-1}. Thus, the mod 3 prime memory scheme reduces the memory contention overhead b to zero. Note that the mod 3 address calculations are performed with special ROMs that produce the mod 3 address and the memory module number for a given address.

Figure 6.16 shows the network and memory configuration for the FFT computation. In this figure, data memories DMMs 0-3 of memory group 0 store P and Q, and data memories DMMs 4-6 of memory group 1 store P' and Q'. The memory groups 0 and 1 are switched in transition from one pass to the next. Figure 6.16 also shows special buffer memories (BMs). They are used to store P' and Q' after data memories are switched when input data read operations terminate. After all the P's and Q's left in the pipeline at the data memory switching are stored into BMs, they are stored one by one into the previous output data memories (input data memories after the data memory switching). This is possible because one of the three data memories is always free for access, since two of them are used for P and Q at one time. For this purpose, each buffer memory (BM) contains the data address and memory module number as well as data itself. Each free memory module in the input data memory group is checked against the information contained in the two buffer memories. Thus, this data buffering scheme reduces the pass transition overhead c to zero.

With the mod 3 prime memory system and the special data buffering scheme, in the MSP, the number of pipeline cycles $C(n)$ is given by

$$C(n) = a + n \cdot 2^{n-1}. \quad (b = c = 0) \tag{6.14}$$

6.6 MSP Programming

The vector operation in the APU is initiated by the CMF (Call Macro Function) instruction. It is used to execute parameterized macro functions, such as matrix computations, FFT, digital filters, and so forth. Like the LDL and STG instructions, the execution of the CMF instruction ends in reading the parameter list and initiating the vector operation in the APU. The actual vector operation is performed in the APU in parallel with the execution of the subsequent instructions in the CPU. When the vector operation terminates, an interrupt is raised to the CPU. The termination of the vector operation can also be checked by the TST instruction. Figure 6.17 shows the execution of the CMF instruction and the termination checks by the TST instruction.

The execution of the CMF instruction causes a trap to transfer control through the trap handling routine to the CMF routine, whose entry point is shared among all the macro functions implemented with the CMF instruction. The CMF routine receives the parameter list specified by the second operand of the CMF instruction, determines which subroutine shall be invoked, and transfers control to the appropriate subroutine. The invoked subroutine, corresponding to some macro function, gets the parameters, sends the control data generated from the given parameters to the APU to form the desired computation network and to set up the registers such as data counters and address generators, and then activates the vector operation in the APU. Control is returned to the instruction stream following the CMF instruction and its execution is resumed when the vector operation is initiated.

Using the UCC (Unit Control Command) and UCD (Unit Control Data), the CPU controls the vector operation in the APU when a CMF instruction is executed. The APU

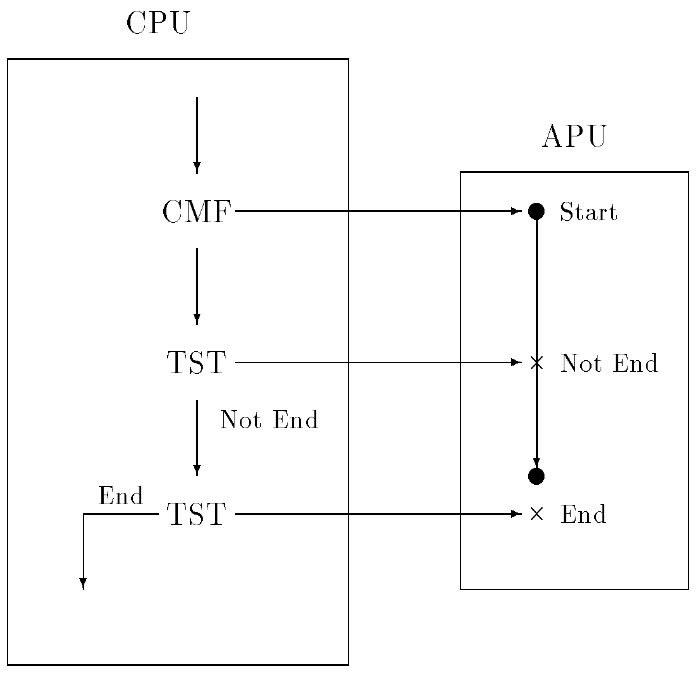
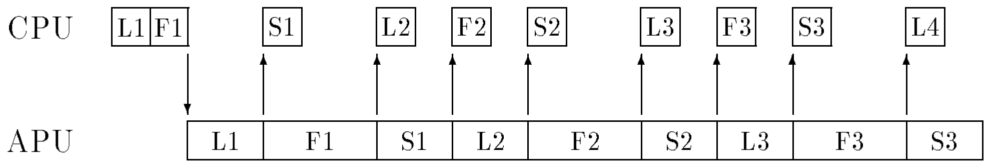


Figure 6.17: The CMF Instruction Execution.



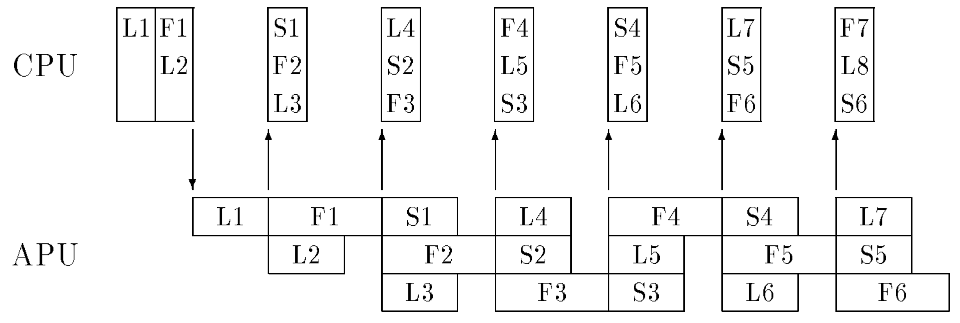
L_i: Load from GMU
F_i: FFT
S_i: Store into GMU

Figure 6.18: Example FFT Processing.

uses a double buffering scheme for control data and can buffer in advance the control data for the next vector operation while holding the current active control data. It reduces the overhead in switching from one vector operation to another, because the CPU can send the control data for the next vector operation before the current operation terminates.

A series of vector operations can be efficiently implemented with an interrupt-based programming technique discussed in the previous chapter. The FFT computation is a good example of the execution of a series of vector operations using the CMF instruction. Consider a task that performs the radix-2 32-point FFT on 64 data sets, each containing 32 sample data. The task contains the CMF instruction to invoke the radix-2 32-point FFT in the APU. The CMF parameter list contains the function number assigned to the radix-2 FFT, the number of points (32), and the number of data sets (64). The function number is used by the CMF routine to find the subroutine for that function. The radix-2 32-point FFT consists of five passes, each including 16 butterfly operations. Sample data are processed, pass by pass, in units of some specified number of data sets, say, eight. Then each pass has $16 \times 8 = 128$ butterflies, and the total number of butterflies for the 64 data sets amounts to $16 \times 8 \times 5 \times 8 = 5120$.

The radix-2 FFT subroutine invoked by the CMF instruction handles 64 data sets in units of eight sets. First, the subroutine sends the control data to transfer the first eight data sets from the GMU to one of the data memory group in the DMU, and initiates the data transfer operation. With the APU's double buffering capability, the CPU sends the control data for the 32-point FFT to be performed on the transferred data sets without waiting for the termination of the data transfer operation for the first data sets. After the CPU finishes sending the control data, control is returned to the instruction stream following the CMF instruction. When the data transfer operation for the first eight data sets terminates, the APU starts the 32-point FFT on the data sets stored in the DMU, using the control data



L$_i$: Load from GMU
F$_i$: FFT
S$_i$: Store into GMU

Figure 6.19: Improved Example FFT Processing.

sent in advance. The termination of the data transfer operation also issues an interrupt to the CPU. The program being executed in the CPU is interrupted, and control is moved to the radix-2 FFT subroutine through the interrupt handling routine. The subroutine prepares the control data to store the FFT results in the DMU into the GMU and returns control to the suspended instruction stream. When the 32-point FFT terminates, an interrupt is issued to the CPU, which suspends the current instruct stream and transfers control to the FFT subroutine. The FFT subroutine sends the control data to the APU to read the second data sets from the GMU into the DMU and then returns control to the suspended instruction stream. The cycle of data load, FFT, and data store operations are repeated until the eight data set groups are processed. This series of operations is illustrated in Figure 6.18.

Since the DMU has four data memory groups that operate in parallel, the above example of FFT computations can be improved with the DMU capability as shown in Figure 6.19, where three operations (loading, FFT, and storing) are scheduled at the same time.

Chapter 7

Discussions

This chapter gives an evaluation of the example implementation of the proposed system architecture, the design methodology of a real-time computing system, the architectural variations, and the related work.

7.1 Evaluation of the Example Implementation

The first version of the MSP system was implemented with over 7,000 standard TTL ICs [133], [134]. In order to evaluate its performance, we used the following three algorithms [134]:

- Complex Multiply and Add
- Radix-2 FFT Butterfly
- 2nd-Order IIR Filter

They are briefly described below. In their description, each data D is assumed to be a 24-bit complex data in the following form:

$$D = D_r + jD_i, \tag{7.1}$$

where D_r is the 12-bit real part and D_i the 12-bit imaginary part.

(1) Complex Multiply and Add

The computation is defined by

$$Z = A \times B + C, \tag{7.2}$$

or

$$\begin{aligned} Z_r &= A_r \times B_r - A_i \times B_i + C_r, && (7.3)\\ Z_i &= A_r \times B_i + A_i \times B_r + C_i. && (7.4) \end{aligned}$$

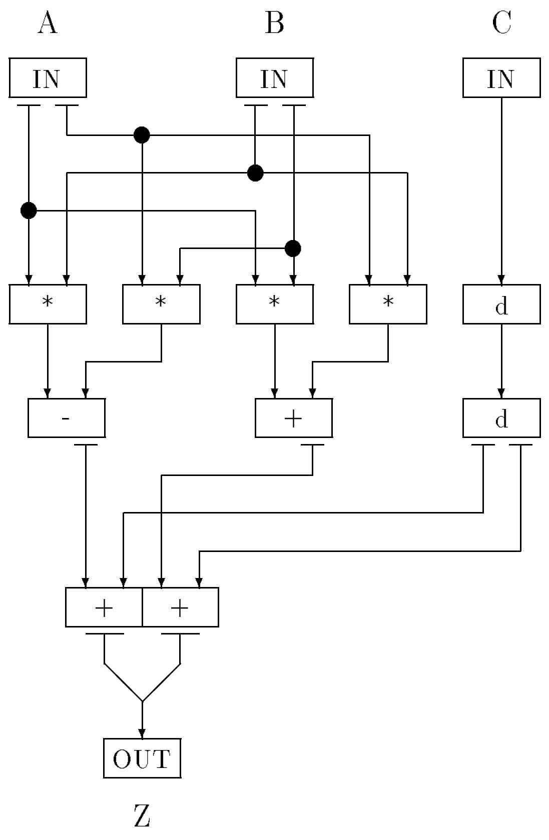


Figure 7.1: Network Configuration for Complex Multiply and Add.

The network configuration for this computation is shown in Figure 7.1.

(2) Radix-2 FFT Butterfly

The radix-2 FFT butterfly computation is defined by

$$\begin{aligned} X &= P + Q \cdot W, & (7.5) \\ Y &= P - Q \cdot W, & (7.6) \end{aligned}$$

or

$$\begin{aligned} X_r &= P_r + R_r, & (7.7) \\ X_i &= P_i + R_i, & (7.8) \\ Y_r &= P_r - R_r, & (7.9) \\ Y_i &= P_i - R_i, & (7.10) \end{aligned}$$

where

$$\begin{aligned} R_r &= Q_r \cdot W_r - Q_i \cdot W_i, & (7.11) \\ R_i &= Q_r \cdot W_i + Q_i \cdot W_r. & (7.12) \end{aligned}$$

The network configuration for this computation is shown in Figure 6.13 in the previous chapter.

(3) 2nd-Order IIR Filter

The original definition of the 2nd-order IIR filter is given by

$$H(z) = \frac{z^2 + A \cdot z + B}{z^2 - C \cdot z - D}. \qquad (7.13)$$

And its computations is defined by

$$\begin{aligned} W_n &= X_n + C \cdot W_{n-1} + D \cdot W_{n-2}, & (7.14) \\ Y_n &= W_n + A \cdot W_{n-1} + B \cdot W_{n-2}. & (7.15) \end{aligned}$$

The network configurations are illustrated in Figure 6.11 in the previous chapter.

In general the performance of a computer is measured in MIPS (Million Instructions Per Second) for general-purpose computers and in MFLOPS (Mega FLoating-point Operations Per Second) for supercomputers. We measure the performance of the MSP by MOPS (Million Operations Per Second). Here we refer to a 12/24-bit addition or 12-bit multiplication as an operation.

The clocking rate in the first version of the MSP was 140 nsec. The value of MOPS can be computed by

$$M = \frac{N}{0.14} \, , \tag{7.16}$$

where N is the number of operations performed at each clock cycle.

In the complex multiply and add computation, 8 operations (4 multiplications and four additions) are performed per each clock cycle. In the FFT butterfly computation, 10 operations (4 multiplications and 6 additions) are performed per each clock cycle. In the 2nd-order IIR filter computation, 4 operations (2 multiplications and 2 additions) are performed per two clock cycles in computing W_n and 4 operations (2 multiplications and 2 additions) per each clock cycle in computing Y_n. Thus the MSP achieved 57 MOPS, 71 MOPS, 14 MOPS, and 22 MOPS in computing the complex multiply and add, the FFT butterfly, W_n in the 2nd-order IIR filter, and Y_n in the same filter, respectively.

The performance of the radix-2 FFT butterfly (71 MOPS) showed that the MSP was 60 times and 3.8 times faster than the IBM 4341-12 (1.2 MIPS) and the IBM 3081K (18.8 MIPS), respectively. This high performance was due to the special FFT hardware described in the previous chapter. Since the clocking rate of the MSP was 140 nsec, it computed one butterfly operation in 140 nsec in the FFT computation. When it performed the 2048-point FFT, which consists of 11,264 butterfly operations, it finished the computation in 1.6 msec. It was the very high performance that the CRAY-1 computer could not achieve at that time, although its basic operations were 64-bit floating-point additions or multiplications, more complicated operations than those performed in the MSP.

We designed the MSP based on the performance models described in Chapter 4. The target signal processing application for the MSP contained a task with a mix of vector computations, including FFT. The vectorizable part of the task was 98%, and the serial part 2%, implying that the theoretical upper bound of the speedup for this task was 50 according to Equation 4.9. The required speedup was roughly 30. From Equation 4.10, the acceleration factor A was given by

$$A = \frac{0.98}{1/30 - 0.02} = 75.4 \; . \tag{7.17}$$

That is, the MSP system had to provide a vector processor that could perform the vectorizable part of the task 75.4 times faster than the existing processor in order to achieve the speedup of 30. We investigated the computing technologies, both hardware and software, to achieve that acceleration factor and found that no software solution could achieve it; we needed a hardware solution. We estimated that a hardware solution could achieve the speedup of 15 with hardware pipeline based on TTL/CMOS technology.

The analysis of the task showed that 90% of the execution time of the vectorizable part of that task was spent for FFT computations. Since each butterfly operation contains 10 arithmetic operations, we could get the speedup of 10 by parallelizing it. The total speedup obtained by parallelization was $1/(0.1 + 0.9/10) = 5.3$. Therefore, the estimated speedup obtained by hardware pipeline and parallelization was $15 \times 5.3 = 79.5 > 75.4$.

The first version of the MSP with the clocking rate of 140 nsec actually achieved the speedup of 11 in hardware pipeline, the acceleration factor of $11 \times 5.3 = 58.3$, and the total speedup of $1/(0.02 + 0.98/58.3) = 27.0$ for the task. Therefore, we recognized that the second version of the MSP should be $75.4/58.3 = 1.3$ times faster in order to achieve the total speedup of 30, implying that its clocking rate would be less than $140/1.3 = 108$ nsec.

The implementation of the APU was fairly straightforward. The subdivision of the computation network reduced the total amount of wiring and the total number of connection points. This was done with the analysis of the operations performed in the target applications. We investigated alternative interconnections of functional modules. However, the bus-structured interconnection employed in the APU was simple and fast, compared to the other alternatives. Although we reduced the number of data buses by dividing the total network into three subnetworks, we still had a wiring problem in the physical design of the APU. It is not just an engineering problem, but it is a fundamental problem, which can make the total system design invalid.

We implemented a special mechanism for FFT on the top of the dynamically reconfigured computation network. Because of that mechanism, the MSP achieved the maximum performance for the FFT computations. Although we added special mod 3 address generators, special data buffers, and other small specials, the basic computational capability for FFT was derived from the basic structure of the computation network. It showed that the computation network is very powerful and general for heavy computations like FFT.

We designed four groups of four data memory modules in the APU. They were very efficient and useful, since memory accesses were critical both in vector pipelining and in system-wide pipelining. The particular numbers of groups and data memory modules in each group were derived from the design requirements for the MSP.

The CPU is based on a custom-made 24-bit processor. We could have used a commercial microprocessor from the functional point of view. From the application viewpoint, however, a custom processor was required. Unfortunately, we didn't choose the RISC architecture for the processor. When the MSP was designed, the RISC architecture was gaining momentum in the computer industry, but we didn't have a chance to evaluate it. If we could try it again, we would choose the RISC architecture for the control processor.

When the MSP was designed for the first time, the assembler was the only programming tool for it. Many signal processing algorithms were implemented with it manually, since the total number of functional modules is not large and the signal processing algorithms were not complicated. However, many programmers complained about the difficulties in programming the APU and debugging their programs. A couple of programming tools were available for those programmers for the later version of the MSP.

7.2 Design Methodology

We have presented the new real-time computing system architecture, called *Dynamically Reconfigurable Architecture* or *DRA* in the preceding chapters. As briefly mentioned in Chapter 1, the way we have presented it reflects a design methodology for a real-time computing system. We have followed the following steps:

(1) to define the target class of real-time applications;
(2) to review the existing architectures;
(3) to study the computing models;
(4) to propose the architecture;
(5) to study the programming aspects of the architecture; and
(6) to implement a prototype of the architecture.

These design steps are fairly standard in designing a computing system in general. As discussed in Chapter 2, however, the many previous real-time computing systems have been designed in an ad hoc manner, based on the a priori knowledge of the applications. Using a priori knowledge of the applications is nothing wrong in designing a suitable architecture. Heavily dependent on it, however, the conventional design has been just a collection of feature modules, each associated with some aspect of the applications, and lacking the notion of resource sharing, leading to an inefficient design, especially for large-scale real-time systems. The conventional design method can be applied to small- and medium-scale real-time systems, because those systems do not require many hardware and software resources anyway. But it cannot be applied to large-scale real-time systems, because the amount of resources required by its design prohibits their reasonable implementation. Resource sharing is necessary to make their implementation reasonable. As discussed in Chapter 2, resource sharing raises problems not only in the functional design but also in the timing design in general.

With the DRA, we have shown that resource sharing can be efficiently incorporated into the architectural design without sacrificing performance and without increasing design complexity. In the DRA, resource sharing is realized by functional modules and their dynamic interconnections. Many functional modules are primitive operation-level modules that can be combined to perform higher-level functions as well as to form one or more pipelines. Functional modules can be functionally combined by making logical connections between them, which automatically form pipelines. This simple basic principle remains the same no matter how many functional modules are involved. The reconfigurability of the network enables functional modules to be reused for different operations. Thus the dynamically reconfigured computation network in the DRA is very simple in structure but very powerful in concept. It can be easily applied to large real-time systems.

Since our target applications were based on vector operations, they were easily implemented on the DRA system discussed in Chapter 6, and the system achieved high performance. One of the excellent features of the computation network is that the configurability of the network for an operation guarantees the performance delivered to the operation. That is, if a computation network can be configured to perform a particular operation, it can achieve the performance as expected, since the network processes data in pipeline without any probabilistic or statistical factor in behavior. We avoided as many probabilistic or statistical factors as possible in the DRA design so that real-time applications could be easily implemented using the DRA without timing problems.

We have presented the execution model upon which the proposed architecture is based in Chapter 4. The proposed architecture described in Chapter 5 is derived from the theoretical execution model, and its design is simple and flexible. The specification of the architecture is not tight and allows many architectural variations to be implemented.

The computation network can be implemented with a variety of functional modules and interconnections. As discussed in Chapter 6, the MSP uses single-cycle functional modules, which operate in one pipeline cycle and contain one staging register in it, since all the computations are performed on 12-bit or 24-bit integer data. The computation network consisting of only single-cycle modules makes the timing design easy, because the number of functional modules on a path from input to output is equal to the number of pipeline stages. The other implementation may use floating-point arithmetic modules, which are multi-cycle modules. The computation network including multi-cycle modules makes the timing design harder and requires some software tool for that.

There are many possible interconnections for functional modules. The more general the interconnection is, the more wiring is required and the more difficult the implementation is, in general. In the MSP, the computation network is divided into three subnetworks in order to reduce the total amount of wiring and the number of connection points. This subdivision of the network was derived from the analysis of the operations required for the target applications. The MSP uses a bus-structured network instead of a multi-stage interconnection network, because the bus-structured network works well for a relatively small number of connection points. If an implementation uses a large number of functional modules, and the computation network includes a large number of connection points, then a multi-stage interconnection network may be a choice. For the computation network using it, the stages in the interconnection network have to be taken into account in the timing design.

The applications in the target class are characterized by a sequence of vector operations on sets of real-time data that move in and out periodically. There may be a small amount of irregular "glue" operations between one vector operation and the succeeding one. Such glue operations, which are performed in the scalar processor (SPU), are usually negligible with respect to execution time; the total execution time is dominated by that of regular vector operations. Parallelism of the operations in the SPU and VPU reduces as much effects of the glue operations as possible on the overall performance of the architecture. If the SPU performance is high, it can control multiple VPUs, leading to the design of the DRA system with multiple VPUs.

7.3 Related Work

The concept of *dynamic reconfiguration* is not new. It has been discussed with fault-tolerant computing and studied as an attribute for computer architecture, interconnection structure, system software, and task scheduling [169]. Dynamic reconfiguration is roughly defined to be the capability of undergoing changes in the semantics or the interconnection of the components in a dynamic way. A major issue in research on the topic is how to dynamically reconfigure the system in an efficient way. Recent research focus has been on general-purpose multiprocessor systems and VLSI systems [35], [159], [48], [56]. In our research work, however, emphasis is on the application of the dynamic reconfiguration concept to real-time computing, not to fault-tolerant computing. The proposed real-time computing system architecture is called the *Dynamically Reconfigurable Architecture* or *DRA*, because of its capability of reorganizing the computation network dynamically for vector computations.

The characteristics and problems in real-time computing, presented in Chapter 2, are mostly shared by J. A. Stankovic [170], [171]. The major computer architectures reviewed in Chapter 3 include much work related to our research work. It is referenced in that chapter. The performance models presented in Chapter 4 are based on the previous work, which is also referenced in the chapter.

There is some work related to the execution model discussed in Chapter 4. K. Hwang and Z. Xu presented a theoretical work on multipipeline networking for vector compound functions in [87]. They developed the discussions similar to those in Chapter 4. Their discussions are more theoretical. They used the **forpipe** loop as a high-level language construct to specify the vector compound functions (VCFs), each of which is a collection of linked scalar operations to be executed repeatedly many times in a looping structure. Their VCF compilation algorithm generates a *program graph* from a **forpipe** loop, and their pipeline networking algorithm converts the generated program graph into a pipeline network. Their graph partitioning algorithm can partition a program graph into subgraphs with the number of operators equal to or less than a certain number of operators. L. M. Ni and K. Hwang discussed the vector reduction techniques for arithmetic pipelines in [130]. D. Bernstein, H. Boral, and R. Y. Pinter discussed the chaining technique to compute vector expression trees in the context of automatic code generation in [19].

Chapter 8

Concluding Remarks

We have proposed a new computing system architecture, called the *Dynamically Reconfigurable Architecture* or *DRA*, for a class of real-time applications. It includes a Vector Processor Unit (VPU) that performs arithmetic operations on vector data with the *dynamically reconfigurable computation network*, which can be dynamically reorganized for pipelined vector computations. We have also described an example implementation of the proposed architecture.

We have presented the design problems of real-time computing systems and characterized the target class of real-time applications, at which the proposed architecture is targeted. We have reviewed the major existing computer architectures and examined their advantages and disadvantages for the target class of applications. We have presented the design concepts for the proposed architecture, including the discussions about performance and execution models.

The major contributions of our research work are as follows.

1. **Architecture Design Methodology**

   We have demonstrated a design methodology for constructing a real-time computing system through this thesis, although it is not complete in the sense that there are still many issues to be addressed. As discussed in Chapter 2, the previous design of a real-time computing system is based on the collection of features, each corresponding to some aspect of a real-time application, and has provided no flexibility to cover a wide variety of applications. Given a target class of real-time applications, we have identified the architectural features and constructed a new archtiecture. Although the target class of applications is somewhat restricted, we have demonstrated that our approach to the design is viable and effective. We believe that our design methodology can apply to a wider class of real-time applications with minor modifications.

2. **Dynamically Reconfigurable Architecture**

We have demonstrated the effectiveness of the *Dynamically Reconfigurable Architecture* or *DRA* for the target class of real-time applications in Chapter 5. We have also shown that the proposed architecture is flexible and scalable for use as a base architecture for the target class of applications. There is no quantitative restrictions on the architecture, implying that there can be many possible implementations of the architecture. It is specifically important for the implementation with VLSI chips which are expected to continue making a rapid progress in density and speed. There are no architectural constraints on the interconnections between the major system components. This leads to the upgradability of the system. For instance, we can replace a slow SPU with a faster SPU based on a state-of-the-art microprocessor, possibly a RISC microprocessor, to have better control over the VPU, or even multiple VPUs.

3. **Dynamically Reconfigurable Computation Network**

   We have demonstrated the effectiveness of the dynamically reconfigurable computation network for vector computations and its controllability in Chapter 5. Its basic operational principle is structurally very simple but conceptually very powerful. The interconnections of functional modules to achieve the desired functionality automatically form a pipeline network.

4. **Programmability of the Proposed Architecture**

   We have shown the programmability of the proposed architecture in Chapter 5. An interrupt-based programming technique is used to control the computation network formed in the VPU. Programs written for a DRA system are executed by the SPU, which controls the VPU for pipelined vector operations.

5. **Theoretical Basis of the Architecture Design**

   We have shown that the proposed architecture is based on theoretical performance and execution models in Chapter 4. Since the proposed architecture is basically the parallel pipeline architecture, it is not a difficult task to predict its performance when it is implemented. The execution model for the proposed architecture gives what is possible and what is not possible with the architecture at the early stage of the system design. Therefore, the major unpredictable (and undesirable) factors can be eliminated at the early design stage. It also gives opportunities for design automation, which has not been addressed in this thesis.

Possible further research work includes:

1. system design tools

   In order to automate the design of a DRA system, we need design tools to compute the types and numbers of funtional modules required for a given set of vector computations and to produce a switching network to interconnect the functional modules for the required pipelines.

2. the execution model

   The execution model we have presented is very basic. More research is required to extend it for a wider class of algorithms.

3. an efficient interconnection of functional modules

   One of the major difficulties in the implementation of the computation network is in the interconnections of the functional modules. More research is required to find an efficient way to interconnect them.

4. multiple VPUs

   Multiple VPUs can tremendously enhance the performance of the system. The system with multiple VPUs is basically a multiprocessor system. It will raise new problems.

5. programming tools

   More research is required to develop programming tools to enhance the programmability of the proposed architecture. Especially, a vectorizing and parallelizing tool will be useful to run many exiting programs on the DRA system.

Acknowledgements

I am grateful to my thesis advisor, Professor Toshiyuki Kitamori, for his strong support of my work and his invaluable advice. I am also grateful to the members of my thesis committee for their fruitful discussions and suggestions.

I would like to express my deep gratitude to Dr. Tohei Nitta, President of Mitsubishi Electric Research Laboratories, Inc. (MERL), for his continuous encouragement. I thank Mr. Laszlo A. Belady, Chairman and Director of MERL, for his continuous support of my thesis work. I also thank all the MERL research staff members, Dr. Hann-Bin Chuang, Dr. Hugh C. Lauer, Dr. Koichiro Mashiko, Dr. Charles Rich, and Dr. Richard C. Waters, for their encouragement, support, and useful feedback. I would also like to thank all the MERL administrative staff members, Ms. Holly Abernethy, Ms. Lydia Ann McIntosh, and Ms. Sandra Staff, who gave me many good suggestions for English expressions in my thesis.

I am grateful to Dr. Eiichi Ohno, Director of the Headquarters R&D of Mitsubishi Electric Corporation (MELCO), Dr. Tamotsu Nomaguchi, Dr. Hitoshi Ogata, and Mr. Muneaki Ogata, Managers of the MELCO Headquarters R&D, for their encouragement and support. I thank Mr. Nobuyuki Iijima, Mr. Teijiro Sakamoto, and Mr. Toshiyuki Hirai, Managers of the MELCO Kamakura Works, who worked with me on the design of the proposed architecture, implemented it, tested it, and improved it. I also thank Mr. Akira Iwase, Dr. Chiyoji Tanaka, and Mr. Tatsuya Mutoh, who gave me much useful advice while I was associated with the MELCO Computer & Information Systems Laboratory.

I thank Ms. Naomi Iwasaki for her efficient administrative work at the MELCO Headquarters R&D. I would like to thank Ms. Yuko Katsuki, who helped me collect materials for my thesis while I was with the MELCO Headquarters R&D. I would like to express my special thanks to Ms. Tomoko Matsuzawa, who often helped me organize my ideas through valuable discussions, while I was associated with the MELCO Computer & Information Systems Laboratory, and encouraged me by e-mail once I moved to the Boston area.

Finally, I would like to thank Dr. Noriko Ohkami, my wife, for her continuous encouragement and support, and Satoshi and Youki, our children, for their humorous conversations at home during my thesis work.

Bibliographic Notes

Real-Time Computing

The IEEE Computer Society has published a tutorial on hard real-time systems [172], which is a collection of published papers on real-time computing. It includes a paper on the next-generation real-time computing systems [170] and a paper on the scheduling algorithms for hard real-time systems [37]. Technical issues in real-time computing are described in [170] and [171]. References [120] and [97] describe fault-tolerant scheduling algorithms for real-time computing.

Signal Processing

Reference [42] is the seminal paper on FFT (Fast Fourier Transform) by J. W. Cooley and J. W. Tukey. References [23] and [24] are books on Fourier Transform and Fast Fourier Transform (FFT), respectively. Reference [176] presents variations of the radix-2 FFT algorithm. Reference [2, Chapter 7, pp.251–276] describes the computational complexity of FFT. Reference [128] is a book on the principles and signal processing techniques for airborne pulsed doppler radars. Reference [29] is a book on signal processing techniques for radar systems.

Transaction Processing

Reference [116] is a paper on the classification of transaction processing systems based on abstracted transaction properties. Reference [76] is a book on performance modeling of transaction processing systems. Reference [123] is a paper on an OLTP (On-Line Transaction Processing) performance prediction tool, based on the performance analysis described in [132].

VLSI and Computing Technologies

The October 1987 issue of *Scientific American* is a special issue on advanced computing technologies, which includes a paper on a general perspective of advanced computing technologies [150], a paper on a perspective of parallel architectures for advanced computing [63], a paper on a perspective of VLSI chips for advanced computing systems [125], a paper on a perspective of parallel programming for advanced computing [69], a paper on a perspective of data-storage technologies for advanced computing [98], a paper on a perspective of advanced use-computer interfaces [59], and a paper on a perspective of computer networks for advanced computing [91].

Dedicated Architectures

Hardware implementations of FFT are described in [20], [71], [43], [3], and [41]. Microprogram-based hardware implementations of FFT are described in [96], [181], [195], and [4]. Recent FFT processor architectures are described in [15] and [162]. Signal processors with the capability of vector processing are described in [22], [28], and [60]. VLSI digital signal processors are described in [113] and [178]. Reference [65] describes a processor dedicated to radar signal processing. Reference [149] describes a reliable computer architecture for real-time aerospace applications. Reference [92] and [166] describe computing systems for industrial process control. Reference [187] describes a fault-tolerant system architecture for commercial transport aircraft control. Reference [31] describes a system architecture for the space shuttle avionics.

Pipeline Architectures

Reference [93] is a book on the pipeline computers. Reference [156] is a survey paper on pipeline architectures. Reference [85] also contains chapters on pipeline architecture. References [9] and [8] describe the pipeline processing techniques in the IBM System/360 Model 91. Reference [45] describes pipeline processing in VAX 8600. Reference [78] describes the CDC STAR-100 architecture. Reference [185] describes the architecture of TI ASC. References [189] and [34] describe the AP-120B. Reference [85, Chapter 4, pp.233-324] describes the architecture of the FPS-164. Reference [93, Chapter 4, pp.134–173] describes the IBM 3838 array processor. The architecture of the Cray-1 is described in [161], [85, Chapter 4, pp.233–324], [80, Chapter 2, pp.68–143], and [121]. The CDC Cyber-205 is described in [85, Chapter 4, pp.233–324], [80, Chapter 2, pp.68–143], and [121]. References [129] and [79] describe the Fujitsu FACOM VP-200. Reference [140] describes the Hitachi HITAC S-810. Reference [94] describes the NEX SX. References [27] and [142] describe the IBM System/370 Vector Facility introduced with the IBM 3090 [179]. Its performance evaluation is presented in [39], [36], [122], and [186].

Parallel Architectures

References [85], [80], [14], [51], [163], [86], and [68] are general references to the MIMD parallel architecture. Reference [175] describes the Sequent Balance system architecture. Reference [192] describes the Encore Multimax system. References [165] and [82] describe the BBN Butterfly Processor. Experience with the Butterfly system is described in [126] and [136]. References [197] and [196] describe the C.mmp experimental parallel architecture, and references [67] and [66] the Cm* parallel architecture, both developed at the Carnegie Mellon University. Experience with C.mmp and Cm* is described in [89]. Reference [188] describes the S-1 system. Reference [103] describes the Burroughs Scientific Processor (BSP). Reference [95] describes the Denelcor Heterogeneous Element Processor (HEP). Reference [70] describes the NYU Ultracomputer. Reference [102] describes the UIUC Cedar system. Reference [151] describes the IBM RP3. Reference [85, Chapter 9, pp.643–731] describes the Cray X-MP. Reference [84] presents discussions about the multiprocessor supercomputers. References [64], [148], and [47] describe the synchronization issues in multiprocessor systems. Reference [46] gives a survey on the synchronization techniques. Reference [152] describes the hot spot contention problem.

Massively Parallel Architectures

References [25] and [81] describe the Illiac IV developed at the University of Illinois. References [154] and [17] describe Goodyear's Massively Parallel Processor. References [77] and [180] describe Thinking Machines' Connection Machine. Reference [57] describes the ICL DAP system. Reference [62] describes the CLIP system. Reference [16] describes the STARAN system. Reference [160] describes the RPA (Reconfigurable Processor Array) system.

Systolic Architectures

The concept of systolic arrays was first presented by H. T. Kung and C. E. Leiserson in [104]. General references to systolic arrays include [105], [115], [61], [106], and [108]. The wavefront arrays are described in [107], [108], and [109]. Reference [118] presents a rigorous approach to characterizing the systolic structures.

VLIW Architectures

References [54] and [55] describes the VLIW architecture. Reference [50] presents the Bulldog compiler for the VLIW architecture. Reference [53] describes the trace scheduling technique for microcode compaction, on which the VLIW compiler is based. Reference [131] reports on the potential parallelism in programs.

Hypercube Architecture

The hypercube architecture is described in [167], [12], and [74].

Data-Flow Architecture

The dataflow architecture is described in [44], [177], [11], and [117]. Reference [124] describes the VAL programming language for the dataflow architecture.

RISC Architecture

The RISC architecture is described in [145], [155], [75], [147], and [183]. Reference [146] made a case for the RISC architecture. Commercial RISC microprocessors are described in [6] (Motolora 88000), [157] (MIPS R3000), and [114] (Hewlett-Packard PA).

Superscalar Architecture

Reference [88] is a book on the superscalar architecture. References [90] and [182] present the studies on the available instruction-level parallelism for the superscalar architecture.

Performance Models

Reference [7] is a paper on the well known Amdahl's Law. Reference [184] describes the effect of serial portion on the parallel performance. Reference [58] presents the taxonomy of computer architectures proposed by M. J. Flynn. Reference [193] presents elaborate discussions about the parallel performance.

Vectorizing Compilers

The basic techniques for vectorizing programs are described in [101], [144], [5], and [190]. The dependence analysis, on which the vectorizing techniques are based, is described in [99, Chapter 2, pp.80–186], [143], [100], [191], [52], and [119]. Commercial vectorizing compilers include [10] (CDC Cycber-205), [199] (Hitachi S-810), [164] (IBM), [174] (FACOM VP), [127] (Convex), and [40] (Unisys).

Debugging Tools

References [18], [13], and [33] present the snapshot approach to debugging parallel programs. References [32], [38], and [112] present the replay approach. Reference [135] presents an approach to unifying debugging and performance tuning for parallel programs.

Execution Models

Reference [87] describes the multipipeline networking techniques. Reference [130] describes the vector-reduction techniques. Reference [19] describes the optimal chaining of vector operations.

Dynamically Reconfigurable Systems

Reference [169] is a technical report on dynamically reconfigurable systems. Reference [35] presents a taxonomy of reconfiguration techniques for fault-tolerant processor arrays. Reference [159] presents the efficient algorithms for reconfiguring processor arrays in the presence of faulty processors and fixed hardware resources. Reference [48] describes a general approach to designing tree-structured multiprocessors with optimal or near-optimal fault tolerance properties. Reference [56] describes the Reconfigurable Arithmetic Processor developed at MIT.

Miscellaneous Topics

Reference [58] describes Flynn's taxonomy of computer architecture. Reference [30] describes the IBM System/370 architecture. Reference [141] describes the IBM System/370 Extended Architecture. Reference [153] describes the IBM ESA/370 architecture. Reference [83] is a book on computer arithmetic. Reference [99] is a book on the structure of computers. Reference [111] is on prime memory systems. References [168], [1], and [194] describe the interconnection networks.

Our Work

References [133], [134], [137], [138], and [139] describe the techniques used in the MSP, an example implementation of the DRA. Reference [178] presents a 32-bit VLSI digital signal processor. Reference [136] describes our experience with Chrysalis on the BBN Butterfly Processor. Reference [135] presents a unified approach to debugging and performance evaluation of parallel programs. Reference [123] is a paper on an OLTP performance prediction tool called SMART. Reference [132] is a paper on the performance analysis of an OLTP system.

Bibliography

[1] G. B. Adams III, D. P. Agrawal, and H. J. Siegel, "Fault-Tolerant Multistage Interconnection Networks," *IEEE Computer*, Vol.20, No.6, June 1987, pp.14–27.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[3] Z. M. Ali, "A High-Speed FFT Processor," *IEEE Transactions on Communications*, Vol.26, No.5, May 1978, pp.690–696.

[4] J. Allen, "Computer Architecture for Signal Processing," *Proceedings of the IEEE*, Vol.30, No.4, April 1975, pp.624–633.

[5] R. Allen and K. Kennedy, "Automatic Transaltion of FORTRAN Programs to Vector Form," *ACM Transactions on Programming Languages and Systems*, Vol.9, No.4, October 1987, pp.491–542.

[6] M. Alsup, "Motorola's 88000 Family Architecture," *IEEE Micro*, Vol.10, No.3, June 1990, pp.48–66.

[7] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *Proceedings of the AFIPS Spring Joint Computer Conference*, 1967, pp.483–485.

[8] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powers, "The IBM System/360 Model 91: Floating Point Execution Unit," *IBM Journal of Research and Development*, Vol.11, No.1, January 1967, pp.34–53.

[9] D. W. Anderson, F. J. Sparacio, and R. M. Tamasulo, "IBM System 360 Model 91: Machine Philosophy and Instruction Handling," *IBM Journal of Research and Development*, Vol.11, No.1, Janurary 1967, pp.8–24.

[10] C. Arnold, "Vector Optimization on the Cyber 205," *Proceedings of the 1983 International Conference on Parallel Processing*, August 1983, pp.530–536.

[11] Arvind and D. E. Culler, "Dataflow Architectures," *in Annual Reviews in Computer Science*, Vol.1, Annual Reviews Inc., 1986, pp.225–253.

[12] W. C. Athas and C. L. Seitz, "Multicomputers: Message-Passing Concurrent Computers," *IEEE Computer*, Vol.21, No.8, August 1988, pp.9–24.

[13] F. Baiardi, N. DeFrancesco, and G. Vaglini, "Development of a Debugger for a Concurrent Language," *IEEE Transactions on Software Engineering*, Vol.12, No.4, April 1986, pp.547–553.

[14] J. L Baer, "A Survey of Some Theoretical Aspects of Multiprocessing," *ACM Computing Surveys*, Vol.5, No.1, March 1973, pp.31–80.

[15] R. K. Bardin and J. D. Sisk, "Optimizing Architectures for Parallel FFT Processing," *Proceedings of the SPIE*, Vol.1154, *Real-Time Signal Processing XII*, August 1989, pp.147–156.

[16] K. E. Batcher, "STARAN Parallel Processor System Hardware," *Proceedings of the National Computer Conference*, 1974, pp.405–410.

[17] K. E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers*, Vol.29, No.9, September 1980, pp.1–9.

[18] P. Bates and J. Wileden, *High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach*, Technical Report COINS 83-29, Department of Computer and Information Sciences, University of Massashusetts, 1983.

[19] D. Bernstein, H. Boral, and R. Y. Pinter, "Optimal Chaining in Expression Trees," *IEEE Transactions on Computers*, Vol.37, No.11, November 1988, pp.1366–1374.

[20] G. D. Bergland, "Fast Fourier Transform Hardware Implementations — An Overview," *IEEE Transactions on Audio Electroacoustics*, Vol.17, No.6, June 1969, pp.104–108.

[21] G. E. Blelloch, *Vector Models for Data-Parallel Computing*, The MIT Press, 1990.

[22] W. C. Booth, "Approaches to Radar Signal Processing," *IEEE Computer*, Vol.16, No.6, June 1983, pp.32–42.

[23] R. N. Bracewell, *The Fourier Transform and Its Applications*, 2nd Ed., McGraw-Hill, 1978.

[24] E. O. Brigham, *The Fast Fourier Transform*, Prentice-Hall, 1974.

[25] W. J. Bouknight, S. A. Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick, "The Illiac IV System," *Proceedings of the IEEE*, Vol.60, No.4, April 1972, pp.369–388.

[26] I. Y. Bucher, "The Computational Speed of Supercomputers," *Proceedings of the 1983 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1983, pp.151–165.

[27] W. Buchholz, "The IBM System/370 Vector Architecture," *IBM Systems Journal*, Vol.25, No.1, 1986, pp.51–62.

[28] R. S. Bucky *et al.*, "Nonlinear Filtering and Array Computation," *IEEE Computer*, Vol.16, No.6, June 1983, pp.51–61.

[29] W. S. Burdic, *Radar Signal Analysis*, Prentice-Hall, 1968.

[30] R. P. Case and A. Padegs, "Architecture of the IBM System/370," *Communications of the ACM*, Vol.21, No.1, January 1978, pp.73–96.

[31] G. D. Carlow, "Architecture of the Space Shuttle Primary Avionics Software System," *Communications of the ACM*, Vol.27, No.9, September 1984, pp.926–936.

[32] R. H. Carver and K. C. Tai, "Reproducible Testing of Concurrent Programs Based on Shated Variables," *Proceedings of the 6th International Conference on Distributed Computing Systems*, 1986, pp.428–433.

[33] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems*, Vol.3, No.1, February 1985, pp.63–75.

[34] A. E. Charlesworth, "An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family," *IEEE Computer*, Vol.14, No.9, September 1981, pp.18–27.

[35] M. Chean and J. A. B. Fortes, "A Taxonomy of Reconfiguration Techniques for Fault-Tolerant Processor Arrays," *IEEE Computer*, Vol.23. No.1, January 1990, pp.55–69.

[36] H. Cheng, "Vector Pipelining, Chaining, and Speed on the IBM 3090 and Cray X-MP," *IEEE Computer*, Vol.22. No.9, September 1989, pp.31–46.

[37] S.-C. Cheng, J. A. Stankovic, and K. Ramamritham, "Scheduling Algorithms for Hard Real-Time Systems — A Brief Survey," *in* J. A. Stankovic and K. Ramamritham (Eds.), *Tutorial: Hard Real-Time Systems*, IEEE Computer Society, 1988, pp.150–173.

[38] S. Y. Chu, *Debugging Distributed Computations in a Nested Atomic Action System*, Technical Report MIT/LCS/TR-327, Laboratory for Computer Science, Massachusetts Institute of Technology, 1984.

[39] R. S. Clark and T. L. Wilson, "Vector System Performance of the IBM 3090," *IBM Systems Journal*, Vol.25, No.1, 1986, pp.63–82.

[40] H. B. Coleman, "The Vectorizing Compiler for the Unisys ISP," *Proceedings of the 3rd International Conference on Supercomputing*, Vol.2, 1988, pp.186–195.

[41] R. A. Collesidis *et al.*, "An Ultra-High Speed FFT Processor," *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, 1980, pp.784–787.

[42] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation*, Vol.19, No.90, April 1965, pp.297–301.

[43] M. J. Corinthios *et al.*, "A Parallel Radix-4 Fast Fourier Transform Computer." *IEEE Transactions on Computers*, Vol.24, No.1, January 1975, pp.80–92.

[44] J. B. Dennis, "Data Flow Supercomputers," *IEEE Computer*, Vol.13, No.11, November 1980, pp.48–56.

[45] J. DeRosa, R. Glackemeyer, and T. Knight, "Design and Implementation of the VAX 8600 Pipeline," *IEEE Computer*, Vol.18, No.5, May 1985, pp.38–48.

[46] A. Dinning, "A Survey of Synchronization Methods for Parallel Computers," *IEEE Computer*, Vol.22, No.7, July 1989, pp.66–77.

[47] M. Dubois and C. Scheurich, "Synchronization, Coherence, and Event Ordering in Multiprocessors," *IEEE Computer*, Vol.21, No.2, February 1988, pp.9–21.

[48] S. Dutt and J. P. Hayes, "On Designing and Reconfiguring k-Fault-Tolerant Tree Architectures," *IEEE Transactions on Computers*, Vol.39, No.4, April 1990, pp.490–503.

[49] D. L. Eager, J. Zahorjan, and E. D. Lazowska, "Speedup Versus Efficiency in Parallel Systems," *IEEE Transactions on Computers*, Vol.38, No.3, March 1989, pp.408–423.

[50] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, The MIT Press, 1986.

[51] P. H. Enslow, Jr., "Multiprocessor Organization — A Survey," *ACM Computing Surveys*, Vol.9, No.1, March 1977, pp.103–129.

[52] J. Ferrante, K. J. Otttenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems*, Vol.9, No.3, July 1987, pp.319–349.

[53] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, Vol.30, No.7, July 1981, pp.478–490.

[54] J. A. Fisher, "Very Long Instruction Word Architectures and the ELI-512," *Proceedings of the 10th Annual International Symposium on Computer Architecture*, June 1983, pp.140–150.

[55] J. A. Fisher, "The VLIW Machine: A Multiprocessor for Compiling Scientific Code," *IEEE Computer*, Vol.17, No.7, July 1984, pp.45–53.

[56] S. Fiske and W. J. Dally, "The Reconfigurable Arithmetic Processor," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, May-June 1988, pp.30–36.

[57] P. M. Flanders, D. J. Hunt, S. F. Reddaway, D. Parkinson, "Efficient High-Speed Computing with the Distributed-Array Processor," *in* D. J. Kuck, D. H. Lawrie, and A. H. Sameh (Eds.), *High Speed Computing and Algortithm Organization*, Academic Press, 1977.

[58] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, Vol.21, No.9, September 1972, pp.948–960.

[59] J. D. Foley, "Interfaces for Advanced Computing," *Scientific American*, October 1987, pp.82–90.

[60] K. S. Forrstrom *et al.*, "Array Processors in Real-Time Flight Simulation," *IEEE Computer*, Vol.16, No.6, June 1983, pp.62–70.

[61] M. J. Foster and H. T. Kung, "The Design of Special-Purpose VLSI Chips," *IEEE Computer*, Vol.13, No.1, January 1980, pp.26–40.

[62] T. J. Fountain and V. Goetcherian, "CLIP4 Parallel Processing System," *IEE Proceedings*, Vol.127, No.5, Part E, September 1980, pp.219–224.

[63] G. C. Fox and P. C. Messina, "Advanced Computer Architectures," *Scientific American*, October 1987, pp.44–52.

[64] D. D. Gajski and J.-K. Peir, "Essential Issues in Multiprocessor Systems," *IEEE Computer*, Vol.18, No.6, June 1985, pp.9–27.

[65] J. R. Gaskill *et al.*, "Multimode Radar Processor," *Proceedings of the SPIE*, Vol.154, Real-Time Signal Processing, 1978, pp.141–149.

[66] E. F. Gehringer, A. K. Jones, and Z. Z. Segall, "The Cm* Testbed," *IEEE Computer*, Vol.15, No.10, October 1982, pp.40–53.

[67] E. F. Gehringer, D. P. Siewiorek, and Z. Segall, *Parallel Processing: The Cm* Experience*, Digital Press, 1987.

[68] E. F. Gehringer, J. Abullarade, and M. H. Gulyn, "A Survey of Commercial Parallel Processing," *ACM SIGARCH Computer Architecture News*, Vol.16, No.4, September 1988, pp.75–107.

[69] D. Gelernter, "Programming for Advanced Computing," *Scientific American*, October 1987, pp.64–71.

[70] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer — Designing an MIMD Shared Memory Parallel Computer," *IEEE Transactions on Computers*, Vol.32, No.2, February 1983, pp.175–189.

[71] H. L. Groginsky *et al.*, "A Pipeline Fast Fourier Transform," *IEEE Transactons on Computers*, Vol.19, No.11, November 1970, pp.1015–1019.

[72] R. Gupta, A. Zorat, and I. V. Ramakrishnan, "Reconfigurable Multipipelines for Vector Supercomputers," *IEEE Transactions on Computers*, Vol.38, No.9, September 1989, pp.1297–1307.

[73] J. J. Hack, "Peak vs. Sustained Performance in Highly Concurrent Vector Machines," *IEEE Computer*, Vol.19, No.9, September 1986, pp.11–19.

[74] J. P. Hayes and T. Mudge, "Hypercube Supercomputers," *Proceedings of the IEEE*, Vol.77, No.12, December 1989, pp.1829–1841.

[75] J. L. Hennessy, "VLSI Processor Architecture," *IEEE Transactions on Computers*, Vol.33, No.12, December 1984, pp.1221–1246.

[76] W. H. Highleyman, *Performance Analysis of Transaction Processing Systems*, Prentice-Hall, 1989.

[77] W. D. Hillis, *The Connection Machine*, The MIT Press, 1985.

[78] R. G. Hintz and D. P. Tate, "Control Data Star-100 Processor Design," *Proceedings, COMPCON Fall 1972*, September 1972, pp.1–4.

[79] T. Hirakuri, A. Tabata, T. Tsuchimoto, and S. Taguchi, "Supercomputer FACOM VP in Parallel Pipeline Processing System Achieving a Machine Cycle of 7.5 ns," Nikkei Electronics, No.314, April 11, 1983, pp.131–155. (in Japanese)

[80] R. W. Hockney and C. R. Jesshope, *Parallel Computers*, Adam Hilger Ltd., 1981.

[81] R. M. Hord, *The Illiac IV: The First Supercomputer*, IEEE Computer Science Press, 1982.

[82] C. D. Howe, "An Overview of the Butterfly GP1000: A Large-Scale Parallel UNIX Computer," *Proceedings of the 3rd International Conference on Supercomputing*, Vo.2, 1988, pp.134–141.

[83] K. Hwang, *Computer Arithmetic: Principles, Architecture, and Design*, John Wiley & Sons, 1979.

[84] K. Hwang, "Multiprocessor Supercomputers for Scientific/Engineering Applications," *IEEE Computer*, Vol.18, No.6, June 1985, pp.57–73.

[85] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1985.

[86] K. Hwang, "Advanced Parallel Processing with Supercomputer Architectures," *Proceedings of the IEEE*, Vol.75, No.10, October 1987, pp.1348–1379.

[87] K. Hwang and Z. Xu, "Multipipeline Networking for Compound Vector Processing," *IEEE Transactions on Computers*, Vol.37, No.1, January 1988, pp.33–47.

[88] M. Johnson, *Superscalar Microprocessor Design*, Prentice-Hall, 1991.

[89] A. K. Jones and P. Schwarz, "Experience Using Multiprocessor Systems — A Status Report," *ACM Computing Surveys*, Vol.12, No.2, June 1980, pp.121–165.

[90] N. P. Jouppi and D. W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp.272–282.

[91] R. E. Kahn, "Networks for Advanced Computing," *Scientific American*, October 1987, pp.128–135.

[92] H. D. Kirrmann and F. Kaufmann, "Poolpo — A Pool of Processors for Process Control Applications," *IEEE Transactions on Computers*, Vol.33, No.10, October 1984, pp.869–878.

[93] P. M. Kogge, *The Archiecture of Pipeline Computers*, McGraw-Hill, 1981.

[94] K. Kokatsu, S. Watanabe, and R. Kondo, “The SX Supercomputer System with Maximum Performance of 1.3 GFLOPS and 6-ns Machine Cycle Time,” Nikkei Electronics, November 19, 1984, pp.237–272. (in Japanese)

[95] J. S. Kowalik (Ed.), *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, The MIT Press, 1985.

[96] G. L. Kratz *et al.*, “A Microprogrammed Approach to Signal Processing,” *IEEE Transactions on Computers*, Vol.23, No.8, August 1974, pp.808–817.

[97] C. M. Krishna and K. G. Shin, “On Scheduling Tasks with a Quick Recovery from Failure,” *IEEE Transactions on Computers*, Vol.35, No.5, May 1986, pp.448–455.

[98] M. H. Kryder, “Data-Storage Technologies for Advanced Computing,” *Scientific American*, October 1987, pp.72–81.

[99] D. J. Kuck, *The Structure of Computers and Computations*, Vol.1, John Wiley & Sons, 1978.

[100] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, “Dependence Graphs and Compiler Optimizations,” *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, January 1981, pp.207–218.

[101] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, “The Structure of an Advanced Retargetable Vectorizer,” *in* K. Hwang (Ed.), *Tutorial on Supercomputers: Design and Applications*, IEEE Press, 1984, pp.163–178.

[102] D. J. Kuck, E. S. Davidson. D. H. Lawrie, and A. H. Sameh, “Parallel Supercomputing Today and the Cedar Approach,” *Science*, Vol.231, February 28, 1986, pp.967–974.

[103] D. J. Kuck and R. A. Stokes, “The Burroughs Scientific Processor (BSP),” *IEEE Transactions on Computers*, Vol.31, No.5, May 1982, pp.363–376.

[104] H. T. Kung and C. E. Leiserson, “Systolic Arrays (for VLSI),” *in* I. S. Duff and G. W. Stewart, *Sparse Matrix Proceedings, 1978*, SIAM, 1979, pp.256–282.

[105] H. T. Kung, “Let’s Design Algorithms for VLSI Systems,” *Proceedings of the CALTECH Conference on VLSI*, January 1979, pp.65–90.

[106] H. T. Kung, “Why Systolic Architectures?” *IEEE Computer*, Vol.15, No.1, January 1982, pp.37–46.

[107] S.-Y. Kung, K. S. Arun, R. J. Gal-Ezer, and D. V. Bhaskar, "Wavefront Array Processor: Language, Architecture, and Applications," *IEEE Transactions on Computers*, Vol.31, No.11, November 1982, pp.1054–1066.

[108] S.-Y. Kung, "On Supercomputing with Systolic/Wavefront Array Processors," *Proceedings of the IEEE*, Vol.72, No.7, July 1984, pp.867–884.

[109] S.-Y. Kung, S. C. Lo, S. N. Jean, and J. N. Hwang, "Wavefront Array Processors — Concept to Implementation," *IEEE Computer*, Vol.20, No.7, July 1987, pp.18–33.

[110] D. H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Transactions on Computers*, Vol.24, No.12, December 1975, pp.1145–1155.

[111] D. H. Lawrie and C. R. Vora, "The Prime Memory System for Array Access," *IEEE Transactions on Computers*, Vol.31, No.5, May 1982, pp.435–442.

[112] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Transactions on Computers*, Vol.36, No.4, April 1987, pp.471–482.

[113] E. A. Lee, "Programmable DSPs: A Brief Overview," *IEE Micro*, Vol.10, No.5, October 1990, pp.14–16.

[114] R. B. Lee, "Precision Architecture," *IEEE Computer*, Vol.22, No.1, January 1989, pp.78–91.

[115] C. E. Leiserson, "Systolic Priority Queues," *Proceedings of the CALTECH Conference on VLSI*, January 1979, pp.199–214.

[116] A. Leff and C. Pu, "A Classification of Transaction Processing Systems," *IEEE Computer*, Vol.24, No.6, June 1991, pp.63–76.

[117] E. J. Lerner, "Data-Flow Architecture," *IEEE Spectrum*, April 1984, pp.57–62.

[118] H. F. Li and R. Jayakumar, "Systolic Structures: A Notation and Characterization," *Journal of Parallel and Distributed Computing*, Vol.3, No.3, September 1986, pp.373–397.

[119] Z. Li, P.-C. Yew, and C.-Q. Zhu, "An Efficient Data Dependence Analysis for Parallelizing Compilers," *IEEE Transactions on Parallel and Distributed Systems*, Vol.1, No.1, January 1990, pp.26–34.

[120] A. L. Liestman and R. H. Campbell, "A Fault-Tolerant Scheduling Problem," *IEEE Transactions on Software Engineering*, Vol.12, No.11, November 1986, pp.1089–1095.

[121] R. D. Levine, “Supercomputers,” *Scientific American*, January 1982, pp.119–125.

[122] B. Liu and N. Strother, “Programming in VS Fortran on the IBM 3090 for Maximum Vector Performance,” *IEEE Computer*, Vol.21, No.6, June 1988, pp.65–76.

[123] T. Matsuzawa, N. Ogawa, T. Ohkami, and T. Noji, “An OLTP Performance Prediction Tool: SMART,” *Proceedings of the 1991 Spring IPS Japan Conference*, Vol.4, March 1991, pp.73–74. (in Japanese)

[124] J. R. McGraw, “The VAL Language: Description and Analysis,” *ACM Transactions on Programming Languages and Systems*, Vol.4, No.1, January 1982, pp.44–82.

[125] J. D. Meindl, “Chips for Advanced Computing,” *Scientific American*, October 1987, pp.54–62.

[126] J. M. Mellor-Crummey, “Experiences with the BBN Butterfly,” *Digest of Papers, COMPCON Spring 88*, February/March 1988, pp.101–104.

[127] R. Mercer, “The CONVEX FORTRAN 5.0 Compiler,” *Proceedings of the 3rd International Conference on Supercomputing*, Vol.2, 1988, pp.164–175.

[128] G. V. Morris, *Airborne Pulsed Doppler Radar*, Artech House, 1988.

[129] K. Miura and K. Uchida, “FACOM Vector Processor VP-100/VP-200,” *Proceedings of the NATO Advanced Research Workshop on High-Speed Computing*, June 1983, Springer-Verlag, pp.20–22.

[130] L. M. Ni and K. Hwang, “Vector-Reduction Techniques for Arithmetic Pipelines,” *IEEE Transactions on Computers*, Vol.34, No.5, May 1985, pp.404–411.

[131] A. Nicolau and J. A. Fisher, “Measuring the Parallelism Available for Very Long Instruction Word Architectures,” *IEEE Transactions on Computers*, Vol.33, No.11, November 1984, pp.968–976.

[132] N. Ogawa, Y. Yamanaga, T. Matsuzawa, T. Ohkami, and T. Noji, “Performance Analysis of OLTP Systems Using the TPC Model,” *Proceedings of the Operating Systems Meeting of the IPS Japan*, March 1991, Paper No.50-2. (in Japanese)

[133] T. Ohkami, “A Signal Processor with a Dynamically Reconfigurable Computation Network,” *Proceedings of the Information Processing Group Meeting of the IEE Japan*, November 25, 1983, Paper No.IP-83-41. (in Japanese)

[134] T. Ohkami, “MSP: A High-Speed Signal Processor with a Dynamically Reconfigurable Computation Network,” *Proceedings of the 1984 International Symposium on Noise and Clutter Rejection in Radars and Imaging Sensors*, October 1984, pp.633–638.

[135] T. Ohkami, “A Unified Approach to Debugging and Performance Evaluation of Parallel Programs,” *Proceedings of the Programming Languages Meeting of the IPS Japan*, February 1988, Paper No.15-1.

[136] T. Ohkami, “Experience with Chrysalis/Butterfly,” *Proceedings of the Operating Systems Meeting of the IPS Japan*, December 1987, Paper No.37-5.

[137] T. Ohkami, N. Iijima, T. Sakamoto, T. Hirai, A. Iwase, and C. Tanaka, “A Dynamically Reconfigurable Computation Network for Flexible and High-Speed Signal Processing,” *Proceedings of the IEEE International Concerence on Circuits and Computers*, September-October 1982, pp.52–55.

[138] T. Ohkami and A. Iwase, “An Architecture of a Small Control Processor for High-Speed Real-Time Signal Processing,” *Digest of Papers, IEEE COMPCON Fall '84*, September 1984, pp.255–262.

[139] T. Ohkami, A. Iwase, and C. Tanaka, “Pipelined FFT Processing,” *Proceedings of the 1982 Fall IPS Japan Conference*, October 1982, pp.161–162. (in Japanese)

[140] T. Okada, K. Kobayashi, T. Kawabe, and S. Nagashima, “Supercomputer HITAC S-810 Featuring 630 MFLOPS (Maximum Performance) and 1G-Byte Semiconductor Extended Storage,” Nikkei Electronics, No.314, April 11, 1983, pp.159–184. (in Japanese)

[141] A. Padegs, “System/370 Extended Architecture: Design Considerations,” *IBM Journal of Research and Development*, Vol.27, No.3, May 1983, pp.198–205.

[142] A. Padegs, B. B. Moore, R. M. Smith, and W. Buchholz, “The IBM System/370 Vector Architecture: Design Considerations,” *IEEE Transactions on Computers*, Vol.37, No.5, May 1988, pp.509–520.

[143] D. A. Padua, D. J. Kuck, and D. H. Lawrie, “High-Speed Multiprocessors and Compilation Techniques,” *IEEE Transactions on Computers*, Vol.29, No.9, September 1980, pp.763–776.

[144] D. A. Padua and M. J. Wolfe, “Advanced Compiler Optimizations for Supercomputers,” *Communications of the ACM*, Vol.29, No.12, December 1986, pp.1184–1201.

[145] D. A. Patterson and C. H. Séquin, "A VLSI RISC," *IEEE Computer*, Vol.15, No.9, September 1982, pp.8–22.

[146] D. A. Patterson and D. R. Ditzel, "The Case for the Reduced Instruction Set Computer," *ACM SIGARCH Computer Architecture News*, Vol.8, No.6, October 15, 1980, pp.25–33.

[147] D. A. Patterson, "Reduced Instruction Set Computers," *Communications of the ACM*, Vol.28, No.1, Janurary 1985, pp.8–21.

[148] P. C. Patton, "Multiprocessors: Architecture and Applications," *IEEE Computer*, Vol.18, No.6, June 1985, pp.29–40.

[149] A. Pedar and V. V. S. Sarma, "Architecture Optimization of Aerospace Computing Systems," *IEEE Transactions on Computers*, Vol.32, No.10, October 1983, pp.911–922.

[150] A. Peled, "The Next Computer Revolution," *Scientific American*, October 1987, pp.34–42.

[151] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Instroduction and Architecture," *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985, pp.764–771.

[152] G. F. Pfister and V. A. Norton, "Hot Spot Contention and Combining in Multistage Interconnection Networks," *IEEE Transactions on Computers*, Vol.34, No.10, October 1985, pp.943–948.

[153] K. E. Plambeck, "Concepts of Enterprise Systems Architecture," *IBM Systems Journal*, Vol.28, No.1, 1989, pp.39–61.

[154] J. L. Potter (Ed.), *The Massively Parallel Processor*, The MIT Press, 1985.

[155] G. Radin, "The 801 Minicomputer," *IBM Journal of Research and Development*, Vo.23, No.3, May 1983, pp.237–246.

[156] C. V. Ramamoorthy and H. F. Li, "Pipeline Architecture," *ACM Computing Surveys*, Vol.9, No.1, March 1977, pp.61–102.

[157] T. Riordan, G. P. Grewal, S. Hsu, J. Kinsel, J. Libby, R. March, M. Mills, P. Ries, and R. Scofield, "System Design Using the MIPS R3000/3010 RISC Chipset," *Digest of Papers, COMPCON Spring 89*, February/March 1989, pp.494–498.

[158] M. G. Rodd, "Real-Time Issues in Distributed Data Bases for Real-Time Control," *Proceedings of the IFAC/IFIP Workshop on Distributed Databases in Real-Time Control*, October 1989, pp.1–7.

[159] V. P. Roychowdhury, J. Bruck, and T. Kailath, "Efficient Algorithms for Reconfiguration in VLSI/WSI Arrays," *IEEE Transactions on Computers*, Vol.39, No.4, April 1990, pp.480–489.

[160] A. Rushton, *Reconfigurable Processor-Array: A Bit-Sliced Parallel Computer*, The MIT Press, 1989.

[161] R. M. Russell, "The CRAY-1 Computer System," *Communications of the ACM*, Vol.21, No.1, January 1978, pp.63–72.

[162] K. Sapiecha and R. Jarocki, "Modular Architecture for High Performance Implementation of the FFT Algorithm," *IEEE Transactions on Computers*, Vol.39, No.12, December 1990, pp.1464–1468.

[163] M. Satyanarayanan, "Commercial Multiprocessing Systems," *IEEE Computer*, Vol.13, No.5, May 1980, pp.75–96.

[164] R. G. Scarborough and H. G. Kolsky, "A Vectorizing Fortran Compiler," *IBM Journal of Research and Development*, Vol.30, No.2, March 1986, pp.163–171.

[165] G. E. Schmidt, "The Butterfly Parallel Processor," *Proceedings of the 2nd International Conference on Supercomputing*, Vol.1, 1987, pp.362–365.

[166] J. D. Schoeffler, "Distributed Computer Systems for Industrial Process Control," *IEEE Computer*, Vol.17, No.2, February 1984, pp.11–18.

[167] C. L. Seitz, "The Cosmic Cube," *Communications of the ACM*, Vol.28, No.1, January 1985, pp.22–33.

[168] H. J. Siegel, "Interconnection Networks for SIMD Machines," *IEEE Computer*, Vol.12, No.6, June 1979, pp.57–65.

[169] V. P. Srini, *Dynamically Reconfigurable Systems Research*, Technical Report UCB/CSD 88/441, Computer Science Division, University of California at Berkeley, August 1988.

[170] J. A. Stankovic, "Real-Time Computing Systems: The Next Generation," *in* J. A. Stankovic and K. Ramamritham (Eds.), *Tutorial: Hard Real-Time Systems*, IEEE Computer Society, 1988, pp.14–37.

[171] J. A. Stankovic, "A Serious Problem for Next-Generation Systems," *IEEE Computer*, Vol.21, No.10, October 1988, pp.10–19.

[172] J. A. Stankovic and K. Ramamritham (Eds.), *Tutorial: Hard Real-Time Systems*, IEEE Computer Society Press, 1988.

[173] H. R. Strong, "Vector Execution of Flow Graphs," *Journal of the ACM*, Vol.30, No.1, January 1983, pp.186–196.

[174] Y. Tanakura, M. Takiuchi, and S. Kamiya, "A Fortran Compiling System That Can Derive the Full Performance of the Supercomputer," *Nikkei Electronics*, No.387, January 27, 1986, pp.201–226. (in Japanese)

[175] S. Thakkar, P. Gifford, and G. Fielland, "The Balance Multiprocessor System," *IEEE Micro*, Vol.8, No.1, February 1988, pp.57–69.

[176] Tran-Thong, "Algebraic Formulation of the Fast Fourier Transform," *IEEE Circuits and Systems*, Vol.3, No.2, June 1981, pp.9–19.

[177] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture," *ACM Computing Reviews*, Vol.14, No.1, March 1982, pp.93–143.

[178] S. Tsujimichi, T. Ohkami, and Y. Shimazu, "A Next-Generation 32-Bit VLSI Signal Processor," *Proceedings of the IEEE-IECEJ-ASJ International Conference on Acoustics, Speech, and Signal Processing*, April 1986, pp.413–416.

[179] S. G. Tucker, "The IBM 3090 System: An Overview," *IBM Systems Journal*, Vol.25, No.1, 1986, pp.4–19.

[180] L. W. Tucker and G. G. Robertson, "Architecture and Applications of the Connection Machine," *IEEE Computer*, Vol.21, No.8, August 1988, pp.26–38.

[181] S. Uchida *et al.*, "Microprogram Control for High-Speed Pipeline Signal Processor," *Transactions of IECE Japan*, Vol.58-D, No.6, June 1975, pp.328–335.

[182] D. W. Wall, "Limits of Instruction-Level Parallelism," *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp.176–188.

[183] P. Wallich, "Toward Simpler, Faster Computers," *IEEE Spectrum*, August 1985, pp.38–45.

[184] W. H. Ware, "The Ultimate Computer," *IEEE Stepctrum*, Vol.9, No.3, March 1972, pp.84–91.

[185] W. J. Watson, "The TI ASC—A Highly Modular and Flexible Super Computer Architecture," *Proceedings of the AFIPS 1972 Fall Joint Computer Conference*, 1972, pp.221–228.

[186] H. Weberpals, "Architectural Approach to the IBM 3090E Vector Performance," *Parallel Computing*, Vol.13, No.1, January 1990, pp.47–59.

[187] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, C. B. Weinstock, "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proceedings of the IEEE*, Vol.66, No.11, November 1978, pp.1240–1255.

[188] L. C. Widdoes, Jr., "The S-1 Project: Developing High-Performance Digital Computers," *Digest of Papers, COMPCON Spring 80*, 1980, pp.282–291.

[189] W. R. Wittmayer, "Array Processor Provides High Throughput Rates," *Computer Design*, March 1978, pp.93–100.

[190] M. Wolfe, "Vector Optimization vs Vectorization," *Journal of Parallel and Distributed Computing*, Vol.5, No.5, October 1988, pp.551–567.

[191] M. Wolfe and U. Banerjee, "Data Dependence and Its Application to Parallel Processing," *International Journal of Parallel Programming*, Vol.16, No.2, April 1987, pp.137–178.

[192] P. Woodbury, A. Wilson, B. Shein, L. Gernter, P. Y. Chen, J. Barttlet, and Z. Aral, "Shared Memory Multiprocessors: The Right Approach to Parallel Processing," *Digest of Papers, COMPCON Spring 89*, February/March 1989, pp.72–80.

[193] J. Worlton, *A Philosophy of Supercomputing*, Technical Report LA-8849-MS, Los Alamos National Laboratory, June 1981.

[194] C.-L. Wu and T.-Y. Feng, *Tutorial: Interconnection Networks for Parallel and Distributed Processing*, IEEE Computer Society Press, 1984.

[195] Y. S. Wu, "Architectural Considerations of a Signal Processor Under Microprogram Control," *Proceedings of the 1972 Spring Joint Computer Conference*, 1972, pp.675–683.

[196] W. A. Wulf and S. P. Harbison, "Reflections in a Pool of Processors — An Experience Report on C.mmp/Hydra," *Proceedings of the National Computer Conference*, 1978, pp.939–951.

[197] W. A. Wulf, R. Levin, and S. P. Harbison, *HYDRA/C.mmp: An Experimental Computer System*, McGraw-Hill, 1981.

[198] S. Yalamanchili and J. K. Aggarwal, "Reconfiguration Strategies for Parallel Architectures," *IEEE Computer*, Vol.18, No.12, December 1985, pp.44–61.

[199] N. Yasumura, Y. Tanaka, Y. Kanada, and A. Aoyama, "Compiling Algorithms and Techniques for the S-810 Vector Processor," *Proceedings of the 1984 International Conference on Parallel Processing*, August 1984, pp.285–290.