Latency-Directed Multithreaded Computation and Its Architectural Support

Dissertation
zur Erlangung des Doktorgrades
des Fachbereichs Informatik
der Universität Hamburg

vorgelegt von

Xiaoming Fan

aus Hangzhou, China

Hamburg, 1993

Genehmigt vom Fachbereich Informatik der Universität Hamburg

auf Antrag von	Prof. Dr. Klaus von der Heide	(Referent)
und	Prof. Dr.-Ing. Rolf-Rainer Grigat	(Korreferent)

Hamburg, den 24. März 1994

Prof. Dr. Walther von Hahn

Der Sprecher des Fachbereichs

Acknowledgments

I want to express my sincere appreciation to the chinese government for providing me a scholarship for my first year stay in Germany. I would like to thank my supervisor Prof. Dr. Klaus von der Heide for getting me started at the Dept. of Computer Science, University of Hamburg, and helping me along. Many people have provided moral and technical support over the past few years. I am especially grateful to Prof. Dr. Klaus Lagemann, the director of the research group "Technical Aspects of Computer Science", who provided me an assistant position during my stay in his research group, so that I could complete my work. Thanks also go to Prof. Dr. Rolf-Rainer Grigat, Dr. Reinhard Rauscher, Manfred Grove, Norman Hendrich, and André Klindworth who all read the draft carefully and gave many constructive comments and criticisms. I also wish to thank our secretary Heike Tewes for her numerous encouragement and help. Finally, I am grateful to my parents and family, especially to my beloved wife Yili who in so many ways provided me her encouragement and support in the whole period of this work.

Abstract

Multithreading is an architectural technique, which allows the *interleaved execution* of multiple instruction streams (threads) on a von Neumann-like processor. Multithreaded architectures combine features from both von Neumann and dataflow architectures, so that the new architecture possesses the simplicity of von Neumann architectures and avoids the complexity and inefficiencies of dataflow architectures. While multithreading is conceptually appealing, little work has been reported on actual performance evaluation, on the architectural advantages and the limitations, and on the requirements of the execution model and its corresponding hardware support. The overall goal of this work is to investigate these issues systematically. This investigation is based on the belief that some fundamental changes in von Neumann architectures are required to build efficient multithreaded architectures.

The work begins with an investigation of performance aspects of conventional pipelined processors. We use a simple performance model to evaluate different mechanisms used in pipelined processors for reducing the performace lost due to pipeline hazards. Multithreading, also as an alternative to conventional pipelining, allows to completely hide pipeline hazards by exploiting parallelism among multiple instruction streams (threads). To understand the potential performance advantage of the multithreading technique, the performance issues of multithreaded architectures have been investigated. Based on the investigations of different multithreading strategies and their performance, a new multithreaded computation model is proposed, called *latency directed multithreaded execution* (LDME). The LDME model separates instructions into two different groups: in one group all instructions have a fixed execution latency (such as arithmetic or logic instructions), and in the other group all instructions have an unpredictable or longer execution latency (such as floating point operations, remote load or write, synchronization instructions etc.). All instructions with unpredictable or longer latency will be split with the technique called *split phase transaction*. We construct all such execution threads in which the result of an instruction with unpredictable latency will never be used by another instruction within the same thread. The consequence of this approach is that all threads are *latency free*, namely after initiation they are executed until completion without any long blocking. Hence, if sufficient parallelism exists in a program, all kinds of latencies (i.e. memory and long execution latency) can be hidden.

A corresponding processor architecture for the LDME model is then proposed. The main part of the LDME processor consists of a multithreaded pipeline and a synchronization processing unit. The multithreaded pipeline exploits the advantages of multithreading, while the synchronization processing unit executes all latency synchronization threads, which are introduced to synchronize parallel normal threads. Like other dataflow and multithreaded processors, the

proposed LDME processor can hide long remote latencies in a multiprocessor environment. To improve the single-thread performance the LDME processor employs the strategy of extracting inter-thread parallelism only when no intra-parallelism exists. The processor also uses the multithreading technique to completely hide effects of pipeline hazards between instructions within the same thread: the dependency information is integrated in each instruction, so that the processor can detect the dependency after the instruction fetch and switches to another active thread if any dependency is detected. Thus, the processor interleaves several active threads during execution and sustains its peak performance if a limited number of active threads reside in the processor, without some related hardware mechanisns used for detecting and resolving such pipeline hazards as in conventional pipelined processors.

The construct and the evaluation of LDME codes is a further issue of this work. To observe the effects of different latencies on the performance of LDME codes, two different cases for storage of operation data are considered. The simulation results are presented. Finally, some perspectives for further research are outlined.

Contents

List of Abbreviations and Symbols

AD: Activation Descriptor, 117

BTB: Branch Target Buffer, 40

CBP: Code block pointer, 118

CLD: Control and load delay, 85

CPI: clock cycles per instruction, 8

DISC: Distributed Instruction Set Computer, 18

DTD: Dynamic thread descriptor, 119

Dfp: Data frame pointer, 117

Dsp: Descriptor frame pointer, 117

QD: Queue descriptor, 117

HEP: Heterogeneous Element Processor, 54

IBA: Instruction base address, 118

IP: Instruction pointer, 103

ISD: I-structure descriptor, 118

LDME: Latency-directed Multithreaded Execution, 92

Lg: Length of a thread, 103

LST: Latency synchronization thread pointer, 119

MPD: Maximum pipeline delay, 85

MPU: Multithreaded processing unit, 135

MSEOC: Switch on every cycle in multiprocessor environments, 61

O(q): Overflow handler, 108

P_RISC: Parallel RISC, 55

PUMA: Puffer Maschine (in german), 57

SB: Scheduling block, 63

Sc: Synchronization count, 103

SOB: Switch on branch, 60

SOEC: Switch on every cycle, 59

SOEI: Switch on explicit instruction, 62

SOD: Switch on discontinuities, 61

SORL: Switch on remote load, 61

SON: Switch on neccessary, 62

SPD: Static pipeline delay, 85

SPU: Synchronization processing unit, 132

STP: Synchronization thread pointer, 118

TD: Thread descriptor, 117

U(q): Underflow handler, 108

VLIW: Very Long Instruction Word, 42

b_p: the number of pipeline bubbles, 76

C_L: communication latency, 64

D_r: the average distance between two remote accesses to the network, 75

L_c: control latency, 25

L_d: data latency, 25

L_s: structure latency, 25

M_f: the fraction of maximum performance, 29

N_{sp}: saturation point, 67

O_L: operation latency, 25

O_p: overhead for context switching due to pipeline hazards, 76

O_r: overhead for context switching due to communication latencies, 76

O_s: overhead for context switching, 65

p_b: the probability of a conditional branch instruction, 28

p_c: the probability that an instruction causes a control latency, 27

p_{cn}: the probability of a correct prediction by using predict not token, 39

p_{ct}: the probability of a correct prediction by using predict token, 40

p_d: the probability that an instruction causes a data latency, 27

p_e: the hazard probability, 28

p_l: the probability that an instruction contains a load, 32

p_{ld}: the probability that a data hazard arises due to a load, 32

p_m: the miss rate of a BTB Cache, 40

p_{nop}: the average probability to fill a delayed slot, 36

p_r: the probability of an instruction to trigger a remote access, 75

p_s: the probability that a conditional branch instruction takes branch, 28

p_u: the probability of a unconditional branch instruction, 28

p_v: the probability of a false prediction, 39

p_{vn}: the probability of a false prediction by using predict not token, 39

p_{vt}: the probability of a false prediction by using predict token, 39

R: the average run length of a thread, 71

R_r: the average distance between two reomte accesses from a thread, 75

S_L: synchronization latency, 64

T_{np}: the time required to perform N instruction sequentially, 24

T_p: the time required to perform N instruction in pipelining, 24

t_p: period of the clock, 24

$U(n)$: pipeline utilization with n active threads, 66

List of Figures

List of Tables

Chapter 1

Introduction

Computer technology has made enormous progress in the past 40 years. The whole history of computer development that we have seen in this period was a history of inventions, excitements, generations, the best and the largest, high performance etc.. Today, with the rapid development of VLSI (Very Large Scale Integration) technology, it is possible to build fast single-chip processors including high-speed floating-point arithmetic. Some *hot* chips have been announced in the past two years, such as DEC's Alpha [DEC 92a], TI's SuperSparc [Bla 92], MIPS's R4000 [Mil 92], HP's PA-RISC 1.1 [Del 92], Motorola's M88110 [Die 92], and so on. All these chips support 64-bit wide data operations and achieve high performance by integrating large data and instruction caches as well as some floating-point functional units on a single chip, which leads to chips containing more than 1 million transistors.

In spite of the enormous progress, more powerful computer systems are always desirable for the following reasons:

- The improvements in hardware technology have led to a repeal of *Grosch's law* that the best price/performance was obtained with the most powerful uniprocessor [Alm 89]. This means that parallel processors, which consist of a collection of processing elements that can communicate and cooperate to solve large problems, are desirable for more powerful computer systems.

- The improvements of the software technology, such as functional and object-oriented programming, require more architectural support. Two novel examples of such machines are dataflow machines supporting the efficient execution of functional programs [Arv 86] and the Message Driven processor [Dal 86] supporting the efficient execution of object-oriented program code.

- Applications in scientific computing, CAD (Computer Aided Design) VLSI design, data base systems, artificial intelligence etc. are demanding order-of-magnitude performance enhancements which may not be archieved only through integrating the advanced technology into the traditional von Neumann architecture model. New methods and innovative architectures are required to fulfill such requirements.

It has become apparent that computer systems in which a large number of processors work in parallel are strongly desirable. Unfortunately, the lessons we learned in more than 40 years of designing von Neumann uniprocessors do not necessarily carry over to multiprocessors. Two fundamental problems encountered in multiprocessors based on von Neumann uniprocessors have been pointed out by Arvind [Arv 87]:

- the inability to tolerate long latencies from remote memory accesses and
- the inability to support efficient synchronization of parallel activities.

As a radical alternative to the von Neumann architecture, various dataflow architectures have been investigated intensively in the past 20 years. In a pure dataflow architecture the problem with *memory latencies* does not exist because there is no *variable* in the dataflow execution, therefore, no variable modification is required. Furthermore, dataflow architectures offer synchronization at a per-instruction level, i.e. each instance of an instruction is an independent task with specific dependence requirements which must be satisfied prior to initiation. Through a special hardware mechanism called *waiting matching unit* all enabled tasks are efficiently detected without any further overhead.

Unfortunately, the experiences with dataflow machines have shown that not all of the distinguishing characteristics of dataflow architectures contribute towards building more powerful computer systems. As Arvind pointed out [Ian 90]:

> *However, the more we thought about building a fast interpreter*[1]*, the more confused we became about dataflow computers. Were dataflow graphs merely a convenient formalism for software development, or had they something to say about hardware organization as well?*

The traditional dataflow architectures have shown their limitations in hardware implementations as well as in software development (In the next chapter we will discuss these issues in detail). Hence, Arvind has suggested an architecture formed on the principles of split transaction I-Structure [Ian 90] memory references in a von Neumann framework, coupled with data

[1] This interpreter refers here to the software interpreter for the MIT's Tagged-Token Dataflow Architecture (TTDA).

driven rescheduling of suspended instructions. This suggestion has led to Iannucci's Dissertation about dataflow/von Neumann hybrid architecture [Ian 88]. Several other similar works have also been reported [Bic 87] [Bue 87][Dai 90].

After intensive research on dataflow architectures, it is now generally accepted by most computer architects that dataflow and von Neumann architectures represent simply the extrema of an architectural continuum. Many now agree that a hybrid von Neumann and dataflow approach with multiple instruction streams per processor is preferable to both pure dataflow as well as to pure von Neumann architectures.

The work presented here is a further study to exploit such issues. The technique we will handle is called *multithreading*, i.e. the *interleaved execution* of multiple instruction streams (threads) on one processor simultaneously. The corresponding processor architectures are called *multithreaded architectures*. Multithreaded architectures combine the features from both von Neumann and dataflow architectures, possess the simplicity of von Neumann architectures, avoid the complexity and inefficiencies of dataflow architectures, and solve the two fundamental problems in multiprocessing pointed out by Arvind efficiently. Thus, multithreaded architectures represent a *trade-off* between the extrema of the architectural continuum.

The recent efforts to research multithreaded processors have been stimulated in designing multiprocessor systems, especially in designing dataflow machines. While multithreading is conceptually appealing, little work has been reported on actual performance evaluation, on the architectural advantages and the limitations, and on the requirements of the execution model and its corresponding hardware support. The overall goal of this work is to study these issues systematically. This investigation is based on the belief that some fundamental changes in von Neumann architectures are required to build efficient multithreaded architectures. Furthermore, a multithreaded architecture should be efficient not only for multiprocessors but also for uniprocessors.

If we look at the internal structure of modern microprocessors a little deeper, we find that they have a common feature, i.e. *they are too complicated in hardware organization.* A large part of their chip area is devoted to certain complicated hardware mechanisms used to handle pipeline hazards. Furthermore, the current solutions to pipeline hazards are far from satisfaction if the instruction pipeline is deepened in order to further increase the maximal work frequency. For examples, DEC's 21064 (Alpha) has a 7 stage integer pipeline and a 10 stage floating point pipeline with a maximum frequency of 150 MHz [DEC 92a]. With a longer pipeline the problems of pipeline hazards also become more difficult to solve. By observing that pipeline hazards are difficult to avoid because of the basic fact that conventional pipelines improve their performance by exploiting the parallelism from a **single instruction stream**, we ask ourselves:

> *Why have we not tried to accept the fact that pipeline hazards in pipelined execution are unavoidable and to find an approach to mask these hazards instead of trying to resolve them with complicated hardware mechanisms*[2].

The answer to this question is multithreading! It is astonishing, however, that little attention has been paid to handle this issue. In designing uniprocessors, especially modern microprocessors, architects have been concentrating on developing the complicated hardware mechanisms used to detect and resolve pipeline hazards. Those pipeline hazards have become more difficult to solve in modern microprocessors such as superscalar processors in which more than one instruction can be issued for execution per clock cycle. This is due to the basic fact that instruction sets of von Neumann machines are traditionally designed with instructions whose execution time is *latency dependent*, which leads to the situations such as *out-of-order-issue* and *out-of-order-completion*. Therefore, more complicated hardware mechanisms must be introduced to handle these issues. On the other hand, in designing multiprocessors, many researchers have been concentrating on mechanisms for handling two fundamental problems in von Neumann based multiprocessors and ignored the necessary changes in the pipeline architectures to further improve the processor performance by masking possible pipeline hazards through multithreading.

Under this consideration, together with other reasons which will be discussed in the following chapters, we will first discuss some basic issues on pipelining and multithreading. After this, a multithreaded execution model called *latency-directed multithreaded execution* (LDME) will be presented, which aims to hide various latencies arising in program execution. At last, the detailed design of a multithreaded processor, which supports the LDME efficiently, will be described.

This thesis is organized as follows:

Chapter 2 first reviews some basic terms of computer architecture and parallel processing. A detailed survey of three different architecture models is then presented, i.e. von Neumann model, dataflow model and distributed instruction architecture model, which aims at giving a feeling of the main features as well as the advantages and disadvantages of these architecture models.

Chapter 3 is devoted to answer the question asked above. First, the features of conventional pipelined processors are reviewed by using a simple performance model in order to evaluate the effects of pipeline hazards to the processor performance. Further, we also examine some important extensions to conventional pipelines, which include Very Long Instruction Word (VLIW), Superscalar and Superpipeline. In comparison to pipelining, the concept of multi-

[2]Actually, the experiences with pipelined processors have shown that the complete avoidance of pipeline hazards is impossible in conventional pipelined processors [McF 86].

threading is then introduced. Some important multithreaded processors developed in recent years are examined.

Chapter 4 concentrates on the performance advantages of multithreaded architectures. Therefore, we classify multithreaded architectures into several different groups according to their strategies of scheduling an enabled thread for execution. Moreover, a realistic multithreaded processor architecture based on a well known RISC processor is constructed at an abstract level and simulated in detail. The simulated performance corresponds to the prediction by the analytical model.

Chapter 5 proposes a multithreaded execution model, which is based on the observation that all unpredictable latencies can degrade processor performance and must be hidden by exploiting the inter-thread parallelism, and the intra-thread parallelism is but used to improve single thread performance. This execution model is hence called *latency-directed multithreaded execution* (LDME). The basic idea of this model is to separate instructions (operations) between those having a deterministic execution time (latency) and those having an undeterministic execution time. All instructions with undeterministic execution time are split, and the corresponding small threads called *latency synchronization threads* are used to synchronize other threads that will use their results. Some software implementation issues are also examined in order to give a feeling how a LDME program will be executed on an associated multithreaded processor.

Chapter 6 presents the detailed design of a multithreaded processor called the LDME processor, which supports the efficient execution of LDME programs. A LDME processor can be used as a uniprocessor. In this case the LDME processor avoids the performance loss due to pipeline hazards by exploiting the inter-thread parallelism and maintains its single thread performance by exploiting intra-thread parallelism. A LDME processor can also be used as a processing element in a multiprocessor and avoids two fundamental problems of latency and synchronization by employing associated mechanisms. The overall structure of a LDME processor consists of a synchronization processing unit used to execute all latency synchronization threads and an multithreaded processing unit used to exploit the performance advantages of the multithreaded execution. The detailed design of the multithreaded pipeline is described, and its corresponding operational behavior will be defined in the hardware description language VHDL.

Chapter 7 demonstrates the benefits of the LDME codes in comparison to the conventional sequential code. To observe the effects of different latencies on the performance of LDME codes, two different cases for storage of operation data are considered: data in the local memory and data in the remote memory. Based on this consideration several LDME codes are constructed and simulated. Their performance as well as some novel features of the LDME codes will be presented.

Chapter 8 is the conclusion of this work. Directions for further activities are also sketched.

Chapter 2

Performance and Parallel Processing

Computer architects have been increasingly convinced of the importance of processor architecture in recent years, especially, those who are concerned with the design parallel machines. It is, therefore, meaningful to examine the concept of processor architecture and its relation to processor performance first.

2.1 Architecture and Requirements

For a long time the term **processor architecture** has been viewed as the description of attributes of a processor, which can be seen by an assembly programmer. This definition refers essentially to the instruction set plus an execution of the instruction set. This narrow sense of processor architecture proved to be invaluable for defining, for example, the characteristics of a family without committing to particular implementation of architectural characteristics and therefore has been extended by many researchers. The most important extensions of the concept **processor architecture** contain the internal structure and the working mode of the processor (**endoarchitecture**[Das 89]) on one hand, which are necessary for the hardware designer, and an abstraction of description of functional and logical features (**exoarchitecture**[Das 89]) on the other hand, which are seen by software developers and programmers. The development until now has shown that a well designed processor must consider both hardware technology, software supports and applications. In this sense, we say the processor architecture is **a study of combination of hardware structures, their software and the application characteristics to optimize the performance of the processor**. This means that the task of an architect (computer designer) is not alone to combine several functional units such as *registers, ALU, control unit* etc. together to execute instructions. The skill of the processor design is thus to select a set of functional modules, which may be hardware, software or a mix of both, and combine them together, so that this combination as a whole system operates efficiently and fulfils the requirements of both hardware technology, software and application characteristics as much as possible.

The first requirement on a new processor is surely its performance. The performance of a processor often refers to some *quantities* such as response time or processing speed. Performance is frequently measured as a rate of some events per second, so that lower time means higher performance[Hen 90]. There is a distinction between performance based on *elapsed time* called *system performance* and that based on *CPU-time* called *CPU-performance*. The system performance is used to refer to elapsed time on an unloaded system, while CPU performance refers to user time the CPU is computing. The latter is more interesting for the architect, because the former includes all activities until the completion of a task, which may consist of disk access, input/output activities, operating system overhead etc. In the following discussion, we will focus on CPU performance.

CPU time for executing a program is often expressed as follows:

$$CPU\ time = \frac{CPU\ clock\ cycles\ for\ a\ program}{Clock\ rate} \tag{2.1}$$

If we know the number of instructions executed – the instruction path length – then we can calculate a useful metric – *clock cycles per instruction* (CPI):

$$CPI = \frac{CPU\ clock\ cycles\ for\ a\ program}{Instruction\ count} \tag{2.2}$$

We can thus rewrite the first expression again:

$$CPU\ time = \frac{Instruction\ count \times CPI}{Clock\ rate} \tag{2.3}$$

The second requirement on a new architecture is the implementation cost. As mentioned in [Hen 90], the main job of an architect is to determine which attributes are important for the new processor, and then to design such a processor which reaches the maximum performance under the cost restriction. A good performance/cost ratio actually corresponds to a good utilization of processor resources during program execution, i.e. processor efficiency. In other words, at the same cost a processor with the higher efficiency means also higher performance than those with lower efficiency.

The further requirement on architecture design is to exploit the internal machine parallelism. This means to exploit all internal functional units which can operate potentially in parallel. A well designed processor has many different components near saturation when it reaches its peak performance, while a poorly designed processor has some single bottleneck when running at maximun speed, and all other functional units are underutilized. The history of computer design has shown us not only the use of fast device in processor design can improve the performance, but also exploitation of internal machine parallelism. Pipelining and parallel processing are two key techniques used to exploit internal machine parallelism. Pipelining overlaps multiple consecutive instructions in execution simultaneously, while parallel processing performs independent instructions (operations) by multiple functional units.

The above three requirements are specially reflected in designing multiprocessor systems, which are built by connecting multiple single processors through some interconnection network. Arvind pointed out two fundamental issues in multiprocessing: *memory latency* and *synchronization* [Arv 87], which are actually the problem of performance lost due to these two problems. This is also the fundamental motivation of this work to study such architecture models which can well fulfil the fundamental requirements above.

To improve the performance further, the architect must also consider algorithms and parallel architecture. Changes to original algorithms, sometimes simple changes, sometimes total new approaches, may lead to great simplification of hardware implementation. Parallel architectures, however, provide a way of using the inexpensive device technology at much higher performance ranges.

2.2 Algorithms and Parallelism

After the specification of a problem, one has to develop an associated algorithm, which is often represented in a high level programming language and then is mapped to the associated instruction set on a physical machine. Some of the terms used in this work are defined as follows:

> **Definition 2.1:**[Gil 81] An **algorithm** is a quadruple (Z, E, A, f), where Z is a set of interim results, E $\subseteq$ Z a set of inputs and A $\subseteq$ Z a set of outputs, f: Z $\rightarrow$ Z a transformation function. The algorithm begins with its initial state ξ_0 and ends with its end state ξ_t. □

The transformation function defines a sequence of transformations which must end after finite number of steps. From the view of programmers, an algorithm of a computation defines a processing run which is organized in form of a program.

> **Definition 2.2: Parallelism** is the opportunity of simultanueous execution of transformations defined in an algorithm. The parallelism rate is the number of simultaneous transformations. □

According to the kinds of processing run, an algorithm can fall into one of two classes of algorithms: *sequential algorithm* or *parallel algorithm*:

Definition 2.3: A **sequential algorithm** contains a transformation function that defines a set of transformations which must be executed sequentially. □

This definition explicates the sequential feature of transformations, namely, at any instant, one and only one transformation can be performed. In contrast, the concept of a parallel algorithm is defined as:

Definition 2.4: A **parallel algorithm** contains a transformation function that defines a partially ordered set of transformations. The partial order is constructed by the dependencies between transformations. □

According to this definition, the partial order defines a subset of transformations at each transformation step, which are independent and therefore can be performed in parallel.

An algorithm is represented in the form of a program which is executed on a computer. To distinguish the parallelism existing in an algorithm and the parallelism exploited by the processor, we introduce the following definitions:

Definition 2.5: **Instruction parallelism** of a program is a measure of the average number of machine instructions that can be executed simultaneously. The instruction parallelism is only determined by dependencies (data and control) between instructions. □

Instruction parallelism is mainly determined by applications, their organisation in algorithms and their mapping strategies (e.g. compiler) to programs which are executed on physical processors. A scientific computation often contains more instruction parallelism than a deeply recursive AI program. A well written program may contain more instruction parallelism which can be exploited to execute in parallel than a strictly sequential program. Further, instruction parallelism is also determined by the processor architecture, because the processor architecture determines the operation latencies of instructions, which influences dependencies between instructions. For this purpose we have the following definition:

Definition 2.6: **Machine parallelism** of a processor is a measure of the ability of the processor to take advantage of the instruction parallelism. Machine parallelism is often measured by the average number of machine instructions that can be fetched and executed simultaneously. □

To achieve high performance, both instruction and machine parallelism must be exploited. Some programs do not have enough instruction parallelism to take advantage of machine parallelism, and some program do not achieve their potential peak performance because of limited machine parallelism. A challenge in designing a new processor is to achieve a good balance between instruction parallelism and machine parallelism in order to fulfill the requirements above.

2.3 Architecture Models

As mentioned, an algorithm can be organized in form of a program either sequentially or in parallel. Correspondingly, one can distinguish sequential processors and parallel processors which execute sequential algorithms and parallel algorithms respectively. Any parallelism existing in a program can only be exploited by corresponding architectural supports.

If we do not consider the details of device technology and hardware structures, the execution of a program can be described by a set of abstractions that provide to the programmer a simplified view of a processor, i.e. a *computational model.* A computational model describes which operations are supported by a processor, when such operations are executed, how data are accessed, where data and instructions are resident etc. Typically, a model provides abstractions for memory, operations and sequencing. Brown listed the following five key attributes that must be specified by a computational model that includes parallelism [Alm 89]:

- The primitive units of computation or basic actions of the computer (the data types and operations defined by the instruction set).
- The definition of address spaces available to the computation (how data are accessed and stored) (*data mechanism*).
- The scheduling units of computation (rules for partitioning and scheduling the problem for computation using the primitive units) (*control mechanism*).
- The modes and patterns of communication among computers working in parallel, so that they can exchange needed information.
- Synchronization mechanisms to ensure that this information arrives at the right time.

where the points 4 and 5 are of main interest for building multiprocessors. In the following we will discuss three important architecture models.

2.3.1 Von Neumann Model

The best-known computational model is called von-Neumann model. The main principles of this model were devised by von Neumann and his colleagues in designing the ENIAC

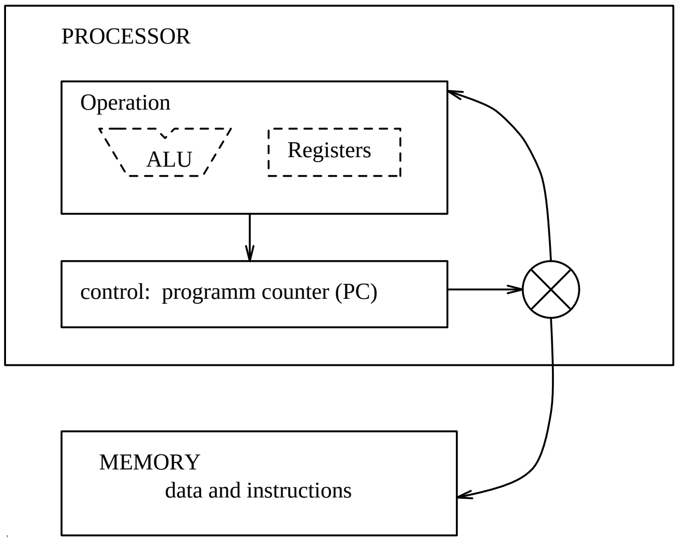


Figure 2.1. The von Neumann computation model

(Electronic Discrete Variable Automatic Computer) in 1946. Figure 2.1 sketches the principle of the von Neumann model.

The essential features of the von Neumann model can be described using the attributes listed above:

- A von Neumann processor executes instructions which contain operations such as "add the contents from two registers and put the result in the third register".
- Data and instructions are stored in the same memory. Each cell of the memory is accessed by an unique address. The content of a cell may be modified during the execution of the program.
- A control scheme fetches one instruction after another from the memory for execution by the processor. The address of the next instruction is given by a central counter, i.e. the program counter. The communication between the processor and the memory is performed *one word at a time.*
- For uniprocessors the communication among instructions are performed through some shared memory cells such as a register file, the main memory etc. The communication between the CPU and I/O devices is performed by some mechanisms such as polling, interrupt, DMA etc.
- The synchronization is defined implicitly in the control scheme. When an instruction is fetched from the memory, this also means that all necessary information is available, because the execution order of instructions must be statically determined.

The features listed above show that a von Neumann processor executes instructions sequentially, namely, the processor can not work faster than the speed of information flow between the processor and the memory through a bidirectional connection. Further, a large amount of the communication through this connection does not consist of useful information, but of addresses or data which are needed to generate such addresses. Because the communication is performed one word at a time, the processing speed of a processor is limited by this connection, which is called **von Neumann bottleneck** by J. Backus [Bac 76].

Backus pointed out further that this bottleneck has greatly influenced the thinking of a programmer: "*Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself but where to find it*". All conventional imperative languages (also called von Neumann languages by Backus) such as *Fortran, Algol, Pascal etc.* contain these features of the von Neumann model. A programmer is forced to think *word-at-a-time* instead of *in terms of larger conceptual units of the task at hand.*

The main advantages of the von Neumann model are its simplicity and its flexibility. The simplicity leads to cost effective implementation which played an important role in early years when the hardware devices were expensive. The flexibility means that von Neumann processors are basically applicable and scalable for *general purpose* problems. This is why von Neumann machines still maintain their position as general-purpose machines until now.

Of cource, most of the current computers have been manifoldly extended from the original von Neumann principles in order to improve performance. Some important extensions are listed as follows:

- separating code and data by introducing different memories (data and instruction memories). Such a processor architecture is often called *Harvard Architecture.*
- introducing memory hierarchy and large word width to reduce the effects of the von Neumann bottleneck.
- introducing overlap in executing instructions and operations, i.e instruction pipelining and operation pipelining.
- exploiting data and instruction parallelism such as VLIW (Very Long Instruction Word), superscalar etc..

These extensions have improved the original model a great deal. In a multiprocessor consisting of multiple von Neumann processors, however, there are further inefficiencies caused by long memory latency and waiting for synchronization events. Latency and synchronization are viewed as two fundamental problems in any von Neumann multiprocessor [Arv 87]. Latency is the elapsed time between making a request and receiving the associated response. Latency is larger in a multiprocessor environment than in a uniprocessor environment because of delay and conflicts in communication network. Synchronisation is the time coordination of parallel activities within a program. Arvind and Iannucci [Arv 87] pointed out:

- Most von Neumann processors are likely to "idle" during long memory references, and such references are unavoidable in parallel machines.
- Waits for synchronization events often require task switching, which is expensive on von Neumann machines. Therefore, only certain types of parallelism can be exploited efficiently.

Even though many mechanisms have been integrated both in parallel programming models and in parallel processor architectures, most of them could not efficiently support synchronization of events, communication of data and naming of global objects, which are essential for efficient execution of a parallel computation model. Software is forced to implement such functions, which leads to large overhead during execution. Indeed, contemporary multiprocessors have task creation and synchronization times in order of hundreds, thousands, and even tens of thousands of instructions [Pap 91a].

2.3.2 Dataflow Model

The dataflow model is a radical alternative to the von Neumann model and can solve the problems mentioned above in general. Dataflow architectures use dataflow graphs as their machine language, which specify only a partial order on the execution of instructions and exploit therefore directly the instruction parallelism. Since the execution of dataflow graphs is *driven* by data values, dataflow architectures provide opportunities for parallel and pipelined execution at the instruction level in a very natural manner. For example, the dataflow graph for the expression

$$(a * b) + (c * d)$$

specifies only that both multiplications must be executed before addition. The multiplications can be executed in any order, even in parallel. This execution model has an obvious advantage when the order in which $a,\ b,\ c,$ and d are generated may not be known at compile time, because an operation is scheduled for execution only when associated operands become available.

A general dataflow architecture model can be sketched as in Figure 2.2. We outline the attributes of the dataflow model as follows:

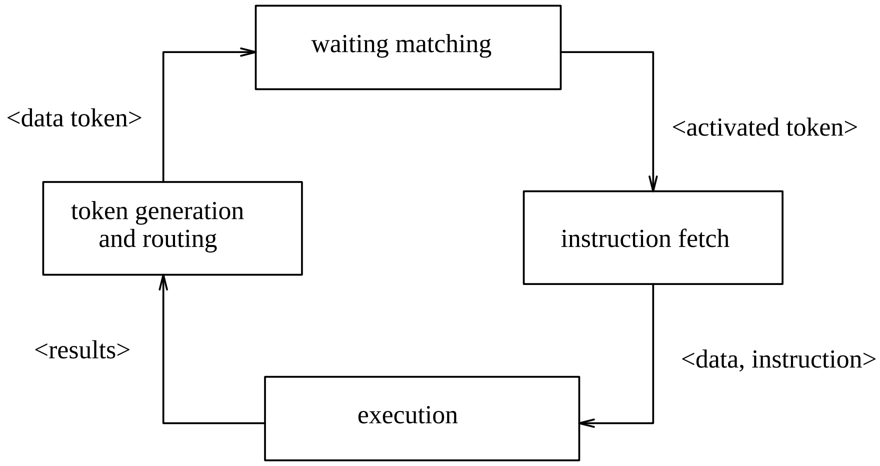


Figure 2.2. The dataflow computation model

- The primitive operations of an activated instruction consists of three parts: receiving input data, executing operation and sending results.

- Values generated by an instruction are transported in form of *data tokens* consisting of data and destination address, i.e a **tag**[1], to one or more successors. The flow of data tokens is determined by data dependencies between instructions. There are no *variables*: old values are absorbed, new values are generated, but no values are modified. This feature is reflected in the semantic of dataflow programming languages, i.e. the *single assignment* rule.

- There is no central control for execution of instructions just like the program counter in the von Neumann model. An instruction becomes executable only when all its operands are available, i.e. all input tokens become available on its input arcs in the dataflow graph.

- Communication between processors is implicit and determined by the map of dataflow graphs to physical processors. The identity of a processor must be contained in each data token.

- Each instruction is an individual task, meaning that all synchronizations are performed at the instruction level. This *low level* synchronzation is realized by the *waiting matching* unit (see Figure 2.2.) which dispatches all instructions whose input data are available to execute.

The essential difference between the dataflow and the von Neumann model lies in the instruction execution mechanism. Using dataflow graphs as machine language, dataflow architectures exploit all potential parallelism of an algorithm in a natural manner.

[1]For dynamic dataflow machines **tag** also contains the identity of a particular firing of the node

The dataflow model avoids the fundamental problems occuring in multiprocessor environments based on von Neumann processors. In a pure dataflow model, the problem *memory latency* does not exist, because there is no *variable* in dataflow execution, therefore, no variable modification is required. In practice, instruction parallelism of data flow graphs can efficiently absorb the communication latency. Synchronization occurs at the individual instruction level and its name space is limited only by the size of **tag**, which is intentionally large. Instruction synchronization is performed by the waiting matching unit and requires no additional overhead such as context switching.

The pure dataflow model suffers from inefficiency of array handling by copying whole array at each array operation. To overcome this inefficiency of array handling, certain structure memory must be integrated into the dataflow architecture. Efficient implementations of arrays in dataflow systems have been studied intensively [Gau 86]. A well known approach proposed by Arvind [Arv 86] to handle array operations is called **I-structure**. An **I-structure** is an array-like data structure, augmented with the kind of synchronization needed for exploiting producer-consumer parallelism without risk of read-write races. In a practical I-structure storage [Arv 90], each location has extra *presence bits*, which indicate the states of the location: *Present, Absent,* or *Waiting*. The special hardware ensures that *Reads* to absent locations are deferred and then resumed after the corresponding *Writes* are performed. In practice, an I-structure is allocated in the global memory space, thus, accesses to the I-structure result in memory latency. An efficient approach to hide such memory latency is called **split phase transaction**, which divides an operation into small parts which separately initiate the operation and later synchronize it prior to using the result. Thus, the latency caused by accessing the structure memory can be hidden by instruction parallelism.

Generally, there are two different implementations of the dataflow model until now: **static** dataflow machines and **dynamic** dataflow machines. In a dynamic dataflow machine data tokens from different invocations of the same code segment are allowed to become available on each arc of dataflow graphs. An instruction is activated only when all input tokens from the same invocation are generated. In a static dataflow machine the dataflow graphs are executed according to the semantic called *one-token-per-arc*, i.e. at any instant only one token is allowed to be available on any arc. Two typical examples are Dennis's static dataflow machine [Den 84] and Arvind's dynamic dataflow machine [Arv 90].

Unfortunately, there is a large *gap* between the dataflow model and efficient hardware supports. Most of the dataflow machines developed so far support *fine grain* parallelism at the instruction level, which leads to many pratical problems both in the implementation and in the applications. This is why dataflow machines have not been accepted after more than twenty years of intensive research. The most important reasons can be summarized as follows:

- High hardware cost in the implementation. The performance of a dataflow machine depends mainly on the size and processing speed of the waiting matching unit which is actually an associative memory and is very expensive to implement. The waiting matching unit is, therefore, often called the *bottleneck* of dataflow machines.

- Inefficient utilization of hardware resources due to a poor mapping of dataflow graphs to processor elements, due to locking parallelism in algorithms or due to the limitation of the waiting matching unit. A dataflow machine typically is made of a long pipeline (30-40 stages), just like the Manchanster Machine [Veen 86]. To fill this long pipeline fully, about 50 parallel instructions are required. A multiprocessor consisting of 20 processor elements requires 1000 parallel instructions to use up the available hardware resources. This means that dataflow machines are very inefficient for applications with limited parallelism.

- The asynchronous execution of each instruction leads to high *fine grain overhead*, which typically is the same order of magnitude over useful work. The overhead mainly stems from additional instructions for loop unrolling, function calls, termination detection, control and structure handling [Arv 88].

- The synchronization of each instruction is unnecessary general, because for a sequential series of instructions the static scheduling by the compiler is more efficient. Moreover, the synchronization at each instruction also leads to high load of the waiting matching unit and therefore influences directly the performance of dataflow machines.

- It is difficult to encode some *critical sections* and *imperative* operations by using pure dataflow graphs [Pap 91b]. Such operations are needed to implement functions of the operating system such as dynamic resource management.

- The asynchronous execution of all instructions also means that it is impossible to schedule instructions statically by the compiler. Such static scheduling often leads to better real time efficiency by both reducing superfluous operand synchronizations and by using rapid registers for communication between instructions.

These inherent features have limited the success of dataflow machines. To overcome such limitation while maintaining the novel features of the dataflow model, many improvements have been done in recent years.

The simplest approach is to enlarge the granularity of the task size in order to reduce synchronization frequency. This method is often called *macro or large grain dataflow* computation. For example, if each task consists of a procedure or a number of sequential instructions, then synchronization occurs only between such *large tasks* instead of each instruction, which may reduce the synchronization frequency dramatically. However, it has been proved that selecting a proper *grain* size and partitioning a program into multiple tasks at this *grain* level is very difficult, and no convincing algorithms have been developed until now [Arv 90]. Moreover, an efficient mechanism for context switching is needed, because a larger task results in a more complex execution context.

The most elegant solution to the waiting matching bottleneck is Papadopoulos and Culler's ETS (Explicit Token Store) model [Cul 91a]. The basic idea of ETS is to introduce a new model of storage which allows the operand matching storage for the execution of a function invocation to be coalesced into an activation frame which is explicitly managed by the compiler, so that the implementation of the waiting matching unit can be realized by using conventional (as opposed to content-addressable) memory technology. Further, the ETS model permits the realization of well-balanced pipelines [Cul 91a].

Incorporating dataflow ideas into the von Neumann model is another important direction. Iannucci proposed a hybrid dataflow/von Neumann architecture [Ian 88], in which the scheduling unit is not each instruction but a sequence of instructions called *scheduling quanta* (SQ). SQs are the largest units within which the decomposing agent, human or algorithm, has direct scheduling control, and the smallest units which the run-time system can manipulate [Ian 88]. Therefore, all SQs can be statically determined at compile-time. The main advantages of the hybrid machine over other dataflow machines lie mainly in two aspects: First such architectures have a better performance/cost ratio, because the instructions within a SQ are executed sequentially which can be realized on a conventional von Neumann processor; second they contain all novel features of a dataflow machine for solving two fundamental problems occuring in multiprocessor environments. Iannucci's work has shown that his hybrid architecture has the same processing power as MIT's TTDA [Arv 90] but is more efficient in the processor architecture.

2.3.3 Distributed Instruction Set Model

The Distributed Instruction Set Computer architecture (DISC) is a further proposal for fine-grain multiprocessing. Using a new parallel instruction set and a distributed control mechanism, DISC explores fine-grained parallel processing in a multiple functional unit system [Wang 91].

A parallel instruction consists of two parts: operation and execution. The operation part contains the normal operation code and operands. The execution part contains the information on data and control dependencies. A software post compiler is used to generate the execution information on data and control dependencies for each instruction. This dependency information is represented in form of a *data tag*, which is used to detect and schedule instructions

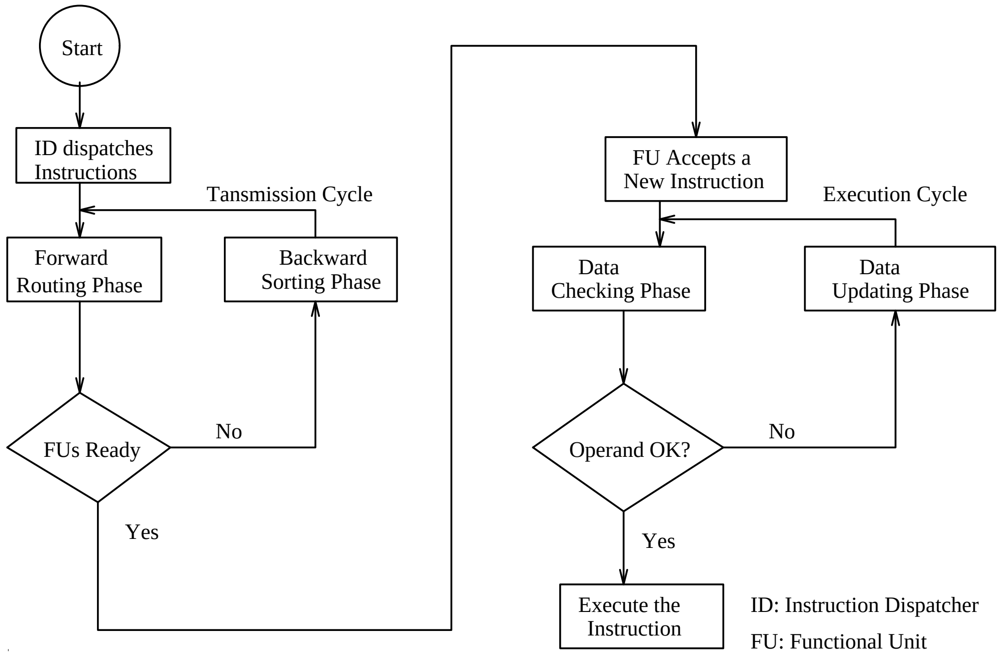


Figure 2.3. DISC control flow chart [Wang 91]

for execution. DISC allows to issue multiple instructions on each cycle. A distributed control mechanism coordinates multiple instruction execution. Each functional unit needs to check the execution information before it proceeds with the execution. If the execution information shows a ready status for the operands, the functional unit proceeds with the execution of the associated instruction. Otherwise, the functional unit updates the execution information and tries to execute the instruction in the next cycle. Figure 2.3 shows the control flow of executing instructions.

DISC can be viewed as a mix of the von Neumann model and the dataflow model. Its distributed control mechanism allows for the execution of multiple instructions on every cycle. Further, an instruction becomes actually executable when its operands are available, which is performed by checking and modifying the execution information. Here we outline only DISC's three main features [Wang 91] instead of listing computational attributes as in the last section:

- Fast multiple instruction issuing mechanism: No decoding work is needed prior to issuing an instruction. Multiple instructions are fetched from the memory and issued directly to multiple functional units.

- Parallel and/or out-of-order execution. No central control mechanism exists for the synchronization of fetched instructions. The data and control dependencies are maintained in a distributed manner among multiple functional units. This minimizes the inter-unit communication and speeds up the overall execution rate.

- Post compiling of programs: The execution information of each instruction is generated statically by a post compiler. This leads to flexibility in organizing a program and allows the optimal instruction scheduling. The post compiling corresponds to the generation of data tokens by software in the sense of the dataflow model.

DISC has actually extended another approach called *reservation station* proposed by R. Tomasulo [And 67] by introducing the software contribution in resolving dependencies, while the *reservation station* resolves the data hazards at run time. We will discuss the *reservation station* approach in the context of performance evaluation of pipelined processors in the next chapter.

The distributed instruction set architecture has its inherent problems which are difficult to solve. Three main problems are listed:

- Busy-waiting: As shown in Figure 2.3, the execution of parallel instructions is divided into two phases: transmission and execution. In the transmission phase, multiple instructions are fetched from the memory and then routed to functional units. If an instruction is not executable, this instruction must be backwarded to the instruction dispatcher which tries to send it to one functional unit on the next cycle. The same occurs also in the execution phase where an instruction can be executed only when all its operands become available. This busy waiting in both the transmission phase and the execution phase may consume a great deal of processor resources.

- Constant operation latency: The execution time of each operation must be predictable by software. Otherwise, it is impossible for the post-compiler to generate the execution information for each instruction.

- Limited performance improvement: If the number of functional units in the system grows, the increase of performance of the processor depends mainly on the ability of the post compiler to find as many parallel instructions as possible. Such an optimal compiler is surely difficult to write, because the instruction parallelism in a single instruction stream is limited. An optimal DISC configuration contains 8 to 16 functional units [Wang 91] and reaches a performance improvement over conventional processors about 3 times. This is due to the fact that the performance improvement is mainly limited by the exploitable instruction parallelism in applications and the ability of the post compiler to find potential parallelism. The performance improvement of DISC therefore is similar to one achieved by other approaches such as superscalar or superpipeline.

- Poor hardware utilization: The simulation results in [Wang 91] are not so encouraging. Both memory blocks and functional units are underutilized during program execution. With the optimal configuration consisting of 4 data memory blocks and 8 functional units, the average utilization of the data memories and functional units is below 30% and 50%, respectively.

In general, a DISC processor is a uniprocessor just like VLIW, superscalar or superpipelined processors, which all improve the performance of the program execution by exploiting the instruction parallelism from a single instruction stream. It is not clear whether DISC processors are more cost effective than others. Since DISC processors employ a software mechanism to determine all possible data and control dependencies, it is expected that DISC's control logic for detecting and resolving dependencies is simpler than, for example, superscalar processors in which all such dependencies are determined dynamically at run time.

2.4 Summary and Remark

Performance and costs are always the two most important criterions in developing a new processor architecture. The rapid development of VLSI technology allows to build multiprocessors with cost effective methods in order to further improve performance. The optimal processor architecture design requires a good balance between algorithms, architectual supports and parallel processing. The von Neumann machines dominate the processor architectures until now, because they are best known for us and simple to build. Moreover, von Neumann machines are clearly superior in the execution of long sequential instruction streams. However, the inability to provide cheap, fine grained synchronization has left doubt whether the von Neumann architecture is reasonable to become the basis for building parallel machines. The dataflow model as a radical alternative suffers from the complexity of implementation in hardware and the large execution overhead in software.

The recent development of hybrid von Neumann/dataflow architecture has shown a new way to build efficient processors. Many extensions to the von Neumann architecture such as pipeline, VLIW etc. have improved the performance a great deal. If we could extend the von Neumann architecture further, so that the new architecture maintains the main features of the von Neumann machines such as simplicity and flexibility, but is also able to avoid their weaknesses, especially when the new architecture is used to build multiprocessor, then it is expected that such an architecture may achieve a higher performance than any von Neumann architecture and be more cost effective than any conventional dataflow machines or their variations.

In this work, we will use a pipelined processor as the base processor and extend it to be a processor which can avoid the weaknesses of conventional pipelined processors and fulfill the requirements mentioned above. Such an extended pipelined processor supporting the execution of multiple instruction streams is called a **multithreaded processor**.

Chapter 3

Pipelining and Multithreading

Pipelining is an implementation technique to improve the performance of sequential von Neumann machines by overlapping execution of multiple consecutive instructions simultaneously. Pipelining has become the key technique used in all high performance processors. In this chapter we will first review the basic principles of pipelined processors, the advantages of this technique, its problems and related solutions. We will present a performance analysis model for conventional pipelined processors and outline the most recent developments (extensions) based on conventional pipelining. The main goal of this review is to give some indication that a more efficient alternative to conventional pipelined execution is strongly desirable in order to handle the new problems resulting from the greater requirements on the machine performance. **Multithreading** is such an alternative. Therefore, the basic principles of multithreading, its main advantages and its penalties are the second issue of this chapter.

3.1 An Overview of Pipelining Principles

The **pipelining** technique in a processor is similar to the assembly line which can be seen in most of modern industry factories. In a computer with pipelining a machine operation (or a function) is split into smaller pieces and separate hardware, termed a *pipeline stage* is allocated to each piece. The pipeline stages are connected one to the next to form a *pipe*. Instructions or data flow through the stages at a synchronous clock, so that multiple instructions can be overlapped in execution simultaneously. We call the whole of this proceeding unit a **pipeline**. The rate at which new entries may be fed to the input of the pipeline depends on many factors which will be described in the next section.

Figure 3.1 shows the structure of a synchronous pipeline. Let t_i denote the time for the logic to compute the suboperation of the stage i, and w the time for the latch to accept results, then the *period of the clock* is defined as:

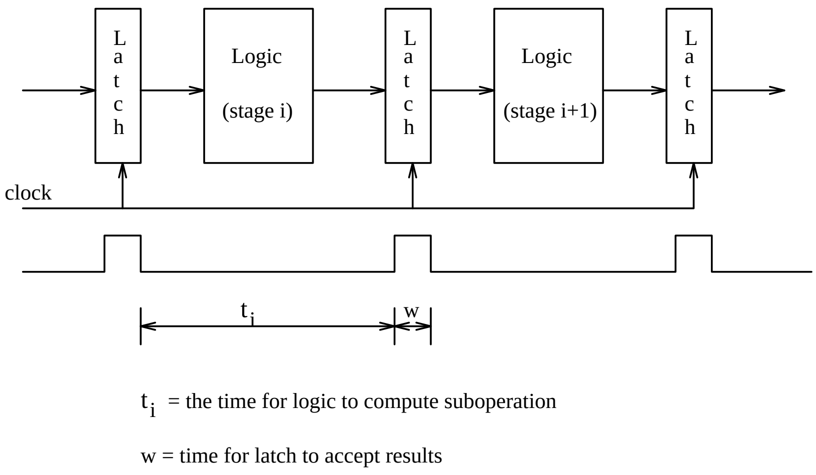


Figure 3.1. The basic pipeline stages

$$t_p = max(t_i) + w, \ i = 1, ..., n \tag{3.1}$$

Assuming a new entry (instruction) can be fed to the pipeline on each cycle, then the time required to perform N instructions is given by

$$T_p = nt_p + (N-1)t_p \tag{3.2}$$

The first term in this expression is the *set-up* time required for the first instruction to propagate through the pipeline. The second term is the time required to stream the remaining $N-1$ instructions through the pipeline.

The performance advantage of pipelining over nonpipelining is obvious. The speed-up over the nonpipelined operation is then

$$\frac{T_{np}}{T_p} = \frac{Nnt_p}{nt_p + (N-1)t_p} = \frac{Nn}{N+n-1} \tag{3.3}$$

and

$$\lim_{N \to \infty} \frac{T_{np}}{T_p} = n \tag{3.4}$$

The equations above represent a simplification of pipelined operation. The speed-up over nonpipelined operation can be achieved only under the assumption that a new instruction can be fed to the pipeline on each cycle. In practice, possible dependencies between instructions which will be overlapped in execution prevent new instructions from entering the pipeline on every clock, which degrades the performance of the pipeline. These cases called *pipeline hazards* are defined as:

Definition 3.1: A **pipeline hazard** is a situation that prevents the next instruction in the instruction stream from execution during its designated clock cycle. There are three kinds of pipeline hazards:

- **Structure hazards** arise from resource conflicts when the hardware cannot support all possible combinations of instructions overlapped in execution simultaneously.
- **Data hazards** arise when an instruction has operands which are results of a previous instruction being executed in the pipeline.
- **Control hazards** arise from the pipelining of branches and other instructions that change the program counter PC.

□

As mentioned, pipeline hazards result in discontinuities of proceeding instructions. An instruction which uses the result of a previous instruction must be delayed until the result of the previous instruction has been produced, namely, it is often necessary to stall the pipeline until some pipeline hazard has been resolved. We describe this situation through the following definition.

Definition 3.2: The **operation latency** O_L is the time which must be delayed until the next instruction in the instruction stream can be issued for execution. The **operation latency** is measured in machine clock cycles. There are three kinds of operation latencies which correspond to the three kinds of pipeline hazards

- **Structure latency** L_s is the latency due to structure hazards.
- **Data latency** L_d is the latency due to data hazards.
- **Control latency** L_c is the latency due to control hazards

□

In general, a pipeline is capable of evaluating several different kinds of functions, which may require different operation time. This means that it is not necessary that an instruction must propagate through all possible pipeline stages. Therefore, a more general pipeline should allow instructions to propagate through different stages in execution. Under this consideration and the definition 3.2 we can define a pipeline generally to be a directed graph as follows:

Definition 3.3: A **pipeline** is a triple $< Sn, Dp, Ol >$, where Sn is a set of function nodes, each performing an atomic operation of instruction execution; Dp is a set of directed arcs connecting the function nodes in Sn to form the data paths; and Ol is a set of directed arcs connecting function nodes to form the dependent paths which represent possible operation latencies.

□

It is assumed that the exact data path of each instruction (operation) is known. Instructions propagate through the Dp arcs, whereas Ol arcs determine possible dependencies of instructions in the pipeline with their succesive instruction. All possible dependencies during the execution can be registered in a global *reservation table*. Just like in a data flow graph, a function node will be *fired* only when an instruction enters the input Dp arc of this node and a *dependency token* is available on its input Ol of this node. After operation, the instruction is forwarded to the next stage along its output Dp arc and a dependency token is backwarded to some previous stages, which actually corresponds to modification of some entry in the reservation table.

3.2 A Simple Performance Model for Pipelined Processors

Of the three pipeline hazards, the structure hazards due to resource conflicts are often easily removed by duplicating the troublesome resource. To simplify our analysis we eliminate effects of all structure hazards. Further, we assume that our target pipeline has a fixed number of stages, and all instructions propagate through each stage of the pipeline. Moreover, the pipeline accepts at most one instruction on each clock cycle. This excludes VLIW or superscalar execution models which will be described in the next section.

As known, on every processor the time taken to complete a program is dominated by three factors (see the expression 2.3):

- the number of instructions required to execute the program,
- the average number of processor cycles required per instruction CPI, and
- the processor cycle time,

namely,

$$\text{CPU-time} = \text{number of instructions} \times \text{CPI} \times \text{clock cycle time}$$

Processor performance is improved by reducing the time taken, which dictates reducing one or more of these factors. The first factor depends on the processor architecture such as instruction set, on applications and on compiler technology. The third factor depends on the device technology and implementation techniques. Pipelining exploits parallelism among the instructions in a sequential instruction stream, which reduces the average number of required processor cycles to execute one instruction, i.e. the second factor CPI. In general, CPI is also determined by many other factors such as the organisation of the pipeline, the instruction set and program structures, all of which contribute to pipeline hazards which increase CPI. Therefore, for pipelined processors CPI is a good performance metric.

Pipeline hazards delay the execution of the next instructions in the instruction stream. In our further analysis, we assume the instruction issue policy *in-order-issue with in-order-completion*, namely, issue instructions in exact program order and write their results in the same order. This is consistent with the instruction issue policy used in most RISC processors with a regular pipeline structure. An instruction can be issued for execution only when no pipeline hazard exists in the pipeline. This also means that only one pipeline hazard may exist in the pipeline at any time.

Let p_d and p_c be defined as:

- p_d - the probability that an instruction causes a data latency
- p_c - the probability that an instruction causes a control latency[1]

After the *set-up* time one instruction will be fetched to execute if no hazard is detected in the pipeline. For an instruction with operation latency, one subsequential instruction must be delayed to be issued until the related hazard is resolved. It is obvious that CPI can be given by

$$\begin{aligned} CPI &= (1 - p_d - p_c)(1) + p_c(L_c + 1) + p_d(L_d + 1) \\ &= 1 + p_c L_c + p_d L_d \end{aligned} \tag{3.5}$$

The expression above only represents the performance of a simplified pipeline. L_c and L_d are the maximum control and data latencies respectively. We do not consider some special situations such as *exception* and *interrupt*. Thus all possible control hazards are generated only by branch instructions. For conditional branch instructions control hazards arise only when branches are taken. For all unconditional branch instructions control hazards always arise, because the branch destination (address) is calculated after decoding the instruction.

[1] In the later discussion no structure hazards will be considered.

Data hazards are generated by those instructions whose results are used as operands by the successive instructions. If the result of instruction i will be used by instruction $i+2$, the stalled cycles, i.e. the data latency, are equal to $L_d - 1$. Therefore, each data manipulation instruction contains a data latency in a range between 0 and L_d. In practice, we should take these situations into account in order to estimate effects of pipeline hazards more accurately. For this purpose we introduce further parameters:

- p_u is the probability that an instruction is a unconditional branch instruction, and
- p_b is the probability that an instruction is a conditional branch instruction,
- p_s is the probability that a conditional branch instruction takes branch

Assuming that the data latency is uniformly distributed between 0 and L_d, then CPI is given by

$$\begin{aligned} CPI &= (1 - p_d - p_b - p_u)(1) + p_u(L_c + 1) + p_b[p_s(L_c + 1) \\ &\quad +(1 - p_s)(1)] + p_d \sum_{i=1}^{L_d} \frac{i}{L_d + 1} \\ &= 1 + (p_u + p_b p_s)L_c + \frac{p_d}{2} L_d \end{aligned} \tag{3.6}$$

If the control latency L_c is equal to the maximum data latency L_d, then the expression above can be rewritten

$$CPI = 1 + p_e L \tag{3.7}$$

where $p_e = p_u + p_b p_s + \frac{p_d}{2}$ and $L = L_d = L_c$. $p_e L$ can thus be viewed as the pipeline stall clock cycles per instruction, which is caused by the pipeline hazard. We call p_e therefore **hazard probability**.

As shown in the expression above, the maximum performance can be achieved only when no pipeline hazards arise in a program or all hazards are resolved by some hardware mechanism, so that one instruction can be fed into the pipeline on each clock cycle. The hazard probability depends mainly on the application. For example, in scientific computations there are often much exploitable data parallelism, and the instruction stream can be ordered in a way that the data dependencies between instructions are avoided by prefetching necessary data sets. Further, control hazards due to branch instructions in such applications can also be minimized, because many branch destinations can be predicted statically. All of these lead to a small hazard probability. Therefore, pipelined processors are basically efficient for such tasks. Note

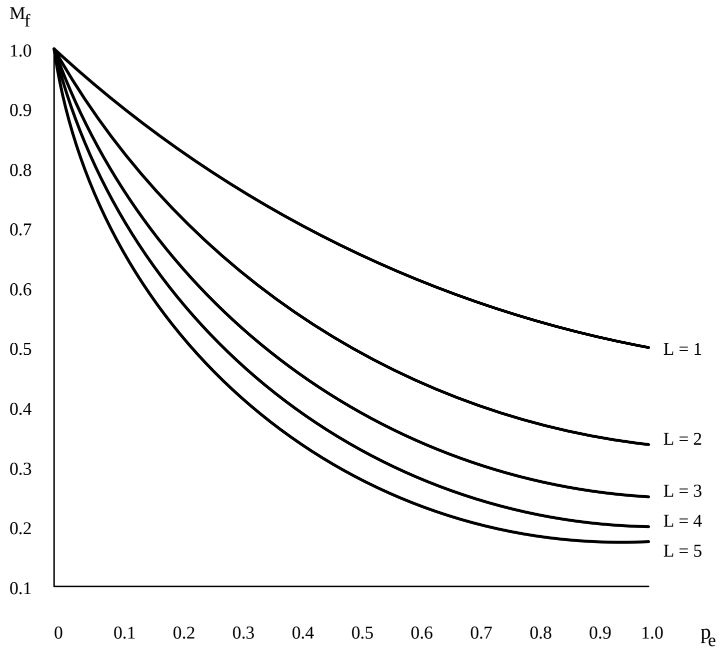


Figure 3.2: Fraction of maximum performance as a function of hazard probability and the pipeline latency

that the hazard probability is partially influenced by the pipeline structure. For example, the probability p_d grows correspondingly when the length of the pipeline increases. The factor L relies mainly on the hardware structure of the pipeline. Many methods or mechanisms were proposed in the past years in order to reduce the value of L, namely, L_d and L_c. We will discuss some of the most important methods in the next sections.

The maximum performance is reached when the hazard penalty becomes zero and one instruction completes every cycle. If we define M_f to be the number of cycles required to complete one instruction at maximum rate divided by the average number of cycles required to execute an instruction, i.e.

$$M_f = \frac{1}{1 + p_e L} = 1/CPI \tag{3.8}$$

we get the fraction of maximum performance realized by the pipeline due to the effects of pipeline hazards. M_f can also be interpreted as the hardware utilization of the pipeline. This means that pipeline hazards lead also to inefficient usage of the hardware resources. Figure 3.2 shows the variations of M_f with 5 different latency lengths.

Example

A typical RISC processor such as Hennessy and Patterson's DLX [Hen 90] contains five pipeline stages:

Data manipulation instruction			Control instruction	
ALU	Load	Store	Branch	Jump
50	20	7	15	8

Tabel 3.1: Distribution of different instruction types (%)

1. IF - instruction fetch
2. ID - instruction decode and register fetch
3. EX- execution and effective address calculation
4. MEM - memory access
5. WB - write back

One typical distribution of executed instructions extracted from the experiments of the DLX architecture [Hen 90] is listed in table 3.1. It is assumed that this basic pipeline employs no additional hardware and software mechanism to reduce the pipeline latency, thus, the effective address of every branch instruction is calculated in the stage MEM. This means that the next instruction after branch can only be fetched for execution when this branch instruction is forwarded to the stage WB. Therefore, the control latency is equal to three processor cycles. Except for branch instruction, all other data manipulation instructions can lead to stall the entering of the next instruction into the pipeline due to data latency. Any data latency can be resolved only when the related instruction has written its result into a register, i.e. after the stage WB. Hence, we have $L = L_d = L_c = 3$. According to experiments in [Hen 90], the average percentage that a conditional branch instruction takes a branch lies between 60% and 70%. Provided $p_s = 65\%$, and $p_d = 50\% + 20\% + 7\%$, then there is a hazard probability of 0.56 and a CPI of 2.68. This means also that M_f is equal to 0.37, namely, the processor reaches only 37% of its maximum performance due to pipeline hazards.

In a processor with a deeper pipeline the hazard probability will grow correspondingly and a smaller fraction of its maximum performance may be achieved if no other mechanisms are added to the pipeline structure, just as shown in Figure 3.2. In the following we will presents some important mechanisms and observe their influences on the performance.

3.3 Detection and Resolution of Pipeline Hazards

In this section we will discuss some important mechanisms used to detect and resolve pipeline hazards. In the next section we will then outline some more recent extensions to the pipelining technique.

3.3.1 Data Hazards

Data hazards arise when the order of access to operands is changed due to pipelining instruction execution in comparison to the order required in the sequential execution. We use $IN(S)$ and $OUT(S)$ to denote the set of inputs (operands) and the set of outputs (results) of the instruction S. In general, data hazards may be classified as one of three types, depending on the order of read and write in the instructions:

> **Definition 3.4:** Given two instructions S_i and S_j, which are fed to the pipeline in the order S_j after S_i. Assuming $x = Out(S_i)$, and if $x \in In(S_j)$ but x will be used by S_j before it is computed by S_i, we say S_j has a **Read-after-Write** hazard to S_i (RAW) and S_j is **flow dependent** on S_i. Assuming $x \in In(S_i)$, and if $x = Out(S_j)$ but x is computed before it is read by S_i, we say S_i has a **Write-after-Read** hazard (WAR) and S_i is **anti-dependent** on S_j. Assuming $x = Out(S_i)$, and if $x = Out(S_j)$ but x will be first computed by S_j, we say S_j has a **Write-after-Write** hazard (WAW) and S_j is **output-dependent** on S_i.
>
> □

Basic pipelining

In the basic DLX pipeline as described in the last section, no WAR and WAW hazards can arise. The operation *read operands* occurs early at the stage 2, while the operation *write result* occurs later at the stage 5, therefore this avoids the WAR hazards. Further, the DLX pipeline writes a register only at the stage 5 (WB) and avoids thus also WAW hazards. Only RAW hazards will arise when the results computed at the stage 3 (EX) are used as operands for the subsequential instructions before they are written into registers at the stage 5 (WB).

In the basic DLX pipeline, such RAW data hazards can be easily resolved by using the technique called **forwarding** (bypassing). The basic idea of this approach is that results generated at the stage EX are directly bypassed to the inputs of the ALU. Associated control logic is responsible for checking whether the forwarded value will be used as operands for the subsequential instructions. The basic bypassing mechanism for the DLX pipeline consists of two ALU result buffers and two multiplexers together with some result bypassing buses. Two ALU result buffers are needed to hold ALU results to be stored into the destination registers in the next two WB stages. For a pipeline with a larger number of stages, more result buffers are needed. Actually, if EW is the number of stages between the ALU stage and the WB stage, then EW result buffers are required to store results temporarily as showed in Figure

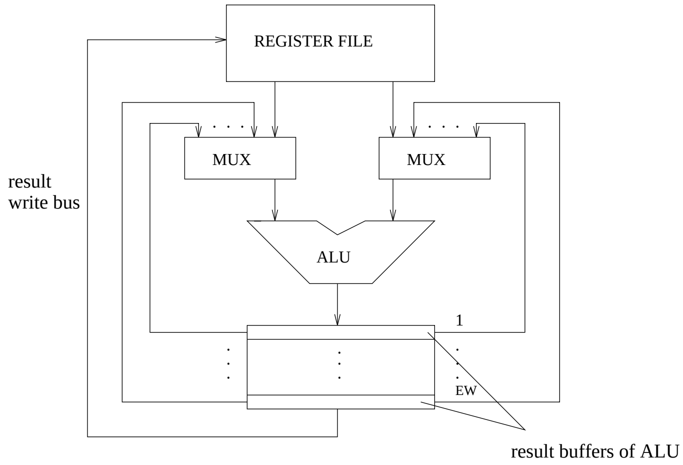


Figure 3.3: The ALU forwarding unit

3.3. Because the ALU operates in a single pipeline stage, no pipeline stall is needed for any combination of ALU instructions if such a bypass mechanism is included.

Another source of data hazards in the DLX pipeline (like in many other pipelined processors) is *load* delay or latency. In the DLX pipeline the data value from the memory is available only at the end of the MEM cycle. If the next instruction uses the value of a load instruction as an operand, the pipeline must be stalled until the fetched value is written into a register. This stall can not be eliminated completely by forwarding alone, which reduces, however, the length of such latencies. A software solution to this problem is called *pipeline scheduling* (instruction scheduling), which reorders the load instructions, so that the results of load instructions are prefetched ahead of the use of these results. A load requiring that the following instruction does not use its result is often called *delayed load*, which corresponds to another concept called *delayed branch* for handling branch: the succesive positions in which other instructions can be inserted and do not cause any stall are called *branch delayed slots*.

We consider this situation in our performance evaluation model and assume two further parameters:

- p_l is the probability that an instruction contains a load,
- p_{ld} is the probability that a data hazard arises due to a load.

The performance form 3.6 can thus be rewritten as follows

$$CPI = 1 + (p_u + p_b p_s)L_c + p_l p_{ld} L_d \qquad (3.9)$$

In this expression only the effects of data hazards due to load instructions are included. All other data hazards are assumed to be resolved by forwarding. Further, the latency of data hazards due to load instructions is also reduced correspondingly by forwarding. For the DLX pipeline the data latency is reduced to 1 by forwarding instead of 3 in the basic pipeline.

Example

The experiments with the DLX processor have shown that about 53% of load instructions will cause data hazards [Hen 90]. Assuming that all other data hazards are resolved by forwarding, the probability p_d becomes $p_l p_{ld} = 0.2 \times 0.53 = 0.106$. If $L_d = 1, L_c = 3, p_u = 8\%, p_b = 15\%$ (from table 3.1) and $p_s = 65\%$, then $CPI = 1.64$, so that $M_f = 0.61$. Using *delayed load* technique, the probability p_{ld} will be reduced further to about 20%, thus CPI is reduced to 1.57, so that M_f becomes 0.64.

As seen, the performance improvement achieved by forwarding is obvious. The fraction of the maximum performance increases from 0.37 to 0.64 at the cost of a limited hardware overhead such as result buffers, multiplexers to ALU inputs, and some logic to control the flow of computed results to the ALU. Therefore, this technique is employed in most pipelined RISC processors.

Advanced pipelining

The solution above is suitable for pipelines in which the length of data latency is constant and data latencies can be predicted statically in software. However, in many pipelined processors, the pipelined operations are more complicated due to the following cases:

- multiple cycle operations. It is typical that different types of floating point operations require a different number of processor cycles to complete their operations. It is not realistic that all floating point operations are completed within one processor cycle,
- data dependencies between different functional units. This is common for most recent RISC processors containing multiple functional units,
- discontinuities of the data stream due to a *cache miss*.

In our analysis, a pipeline containing the features above is called *advanced pipeline* and requires the approach called *dynamic scheduling*, whereby the hardware rearranges the instruction execution to reduce the number of stalls.

Overlapping the execution of instructions with different execution time leads to the situation of *out-of-order-completion* in which an instruction started earlier may complete after other instructions started later. In many pipelined processors, subsequential instructions issued after some stalled instruction due to pipeline hazards are allowed to pass the stalled instruction, which results in another situation: *out-of-order-execution*. These two situations lead to two other data hazards which do not arise in the basic pipeline, i.e. WAR and WAW hazards. A typical example is as follows:

```
R1 : = R1 * R3      (1)
R2 : = R1 + 1       (2)
R1 : = R3 + 1       (3)
R4 : = R1 / R2      (4)
```

In this instruction order the first instruction must be completed before the third instruction, otherwise the fourth instruction will use the result of the first instruction as operand. This means that the result of the third instruction has an *output dependency* on the first instruction, i.e. there is a potential WAW hazard between these two instructions. Further, the third instruction should not complete before the second instruction begins to execute, otherwise the third instruction may overwrite the result of the first instruction, so that the second instruction uses the result of the third instruction instead of the first instruction. This means that the result of the third instruction has an *anti-dependency* on the first operand of the second instruction, i.e. there is a WAR hazard between these two instructions.

In theory, all WAR and WAW can be resolved by the technique called *renaming*. The basic idea of this technique is to introduce additional registers, which are used only to reestablish correspondence between registers and values. The additional registers are allocated dynamically by hardware. Typically, the processor allocates a new register for every new value produced: that is, for every instruction that writes a register. Thus, an instruction identifying the original register obtains the value in the newly allocated register instead. With *renaming*, the example above becomes

```
R1b : = R1a * R3a      (1)
R2a : = R1b + 1        (2)
R1c : = R3a + 1        (3)
R4a : = R1c / R2a      (4)
```

Two other well known methods to resolve such types of data hazards are the *scoreboard* used in the CDC 6600 and Tomasulo's *reservation stations* used in the IBM 360/91. The scoreboard method is based on a data structure where a picture of the data dependencies is constructed and can be modified by the scoreboard on every clock cycle. Besides, the scoreboard also controls whether an instruction can write its result into the destination register. Thus, all data hazard detection and resolution is centralized in the scoreboard. A typical data structure contains the following information: instruction status, functional unit status and register result status. The details about the scoreboard can be found in [Hen 90][Tho 64].

Differently from the scoreboard, Tomasulo's approach to resolve data hazards distributes the similar data structure called *reservation station* to each functional unit instead of a central data structure. Each reservation station contains the decoded instructions waiting for execution in this functional unit and other related information similar to that in the scoreboard. When the value is produced, it is written to any reservation station entry containing a tag for this result. When this result is written into the reservation stations, it may free one or more waiting instructions to be issued by providing a needed input. The main advantages of this approach over the scoreboard are the distribution of detection and resolution logic to each function unit and the elimination of stalls of WAR and WAW hazards through internal renaming registers in each reservation station. A further discussion about the reservation station approach can be found in [Hen 90] [And 67].

Remark:

All of the factors influencing the performance depend mainly on the application and the processor architecture. Further, the complexity of hardware implementation of the dynamic scheduling above prevents these approaches, especially reservation station, from being applied in most of RISC processors. An alternative to the hardware scheduling is to use some software technique in order to extract as much static information from application programs as possible. One general approach is called *software instruction scheduling*: a program or a procedure is divided into many *basic blocks*; all data dependencies within a basic block are statically determined. Together with some control dependencies between instructions, a *dependency graph* can be constructed and will then be scheduled statically by using some heuristic scheduler. The dependencies across basic blocks can also be partially resolved by using, for example, the *trace scheduling* technique [Joh 91]. It should also be noted that such software techniques would show inefficiency when a long pipeline or multiple function units are employed. In some most recent microprocessors long pipelines are employed in order to increase the clock frequency. For example, MIPS's R4000 contains an integer pipeline with 8 pipeline stages and with 3 delayed slots [Mil 92]. An efficient resolution to data hazards is still desired yet.

3.3.2 Control Hazards

Control hazards are another factor which influences the pipeline performance directly. Control hazards occur when instructions changing the program counter are executed. The address of the next instruction after a branch instruction is calculated normally at a later stage. For example, for a conditional branch instruction the new address can be determined only after the condition comparison and the address calculation, while several undesirable subsequential instructions after the branch instruction have been already fetched into the pipeline during this time, which can lead to undesirable effects to the computation. Many methods and mechanisms have been proposed and applied in the design of pipelined processors to solve this problem. The discussion below focuses on three important methods to handle control hazards: **delayed branch, branch prediction** and **branch target buffer**.

Delayed branch

The **delayed branch** approach was already used in the first generation of RISC processors such as Berkeley's RISC-I and RISC-II, Stanford's MIPS and IBM's 801 processors. The basic idea of delayed branch is to find one or more instructions which can be reordered after a branch instruction, so that such instructions are always executed regardless of whether the branch is taken or not.

Assuming L_c is the length of the control hazard, let the instruction i be a branch to the instruction j. Thus, if the branch is taken, the instructions are executed in the sequence $i, i+1, ..., i+L_c, j$, otherwise the sequence of instructions executed is $i, i+1, ..., i+L_c, i+L_c+1$. So we have seen that the instructions $i, i+1, ..., i+L_c$ will always be executed regardless of whether the branch is taken or not. If we can find L_c instructions which are independent of the instruction i, then we call these instructions *useful* instructions. The control hazard is thus resolved by reordering these useful instructions after the branch instruction.

As described, the *delayed branch* is a software technique to handle control hazards. The compiler for a processor is responsible for detecting dependencies in the instruction stream and rearranging the code to insert useful instructions after branch instructions. If no suitable useful instructions could be found, some *no-ops* must be inserted instead. This is true in processors with a length of control hazard more than one instruction. Much work on this topic was done in the Stanford's MIPS project [Gro 88]. An algorithm developed by Gross and Hennessy can fill more than 50% of the first delay slot after a branch instruction with a useful instruction [Hen 90]. It is more difficult to fill the useful instructions for further delay slots.

It should be noted that the effect of the branch probability like the one in the expression 3.6 does not arise here, because the instructions after any branch instruction are always executed. Instead, we must take the effects of *no-ops* into account. Let p_{nop} be the average probability that an arbitrary slot of the L_c delayed slots is filled with no-ops, then p_{nop} is given by

$$p_{nop} = 1 - \frac{p_1 + p_2 + ... + p_{L_c}}{L_c} \tag{3.10}$$

where p_i is the probability that the ith delayed slot is filled with a useful instruction. Thus, the performance expression 3.6 becomes

$$CPI = 1 + (p_u + p_b)p_{nop}L_c + p_d L_d \tag{3.11}$$

In the expression 3.11 no difference of filling useful instructions after conditional instructions and unconditional instructions has been considered. Actually, it is easier to find useful instructions to fill delayed slots after unconditional instructions [Der 87].

Example

In the basic DLX pipeline the length of the control hazard equals to three, because the effective address of the branch destination is only calculated at the stage 4 MEM. Assuming that the first delayed slot is filled with a useful instruction 60% of the time, the second slot only 10% of the time, and the third slot is never used. Using these values for p_i, then $p_{nop} = 0.77$. Assuming also that all data hazards of data manipulation instructions are resolved by *forwarding*, data hazards due to load instructions are reduced by *delayed loading*, and the length of the control hazard $L_c = 3$, then $p_d = 0.04$ and $L_d = 1$. As shown in table 3.1, $p_u = 0.08$ and $p_b = 0.15$. Then $CPI = 1.57$, so that $M_f = 0.64$. To reduce L_c, in the improved DLX pipeline the calculation of branch destination address is moved up at the stage 2 (ID) by using certain associated hardware support [Hen 90], L_c becomes 1, therefore $p_{nop} = 1 - 0.6 = 0.4$, and then $CPI = 1.13$, so that $M_f = 0.88$.

Remark: As seen, the performance of the delayed branch approach is dramatically effected by the length of the control latency. While it is possible for a load/store machine (RISC) with an instruction format that is simple enough to allow the calculation of the branch destination address at the decode stage like the improved DLX pipeline, it is very difficult for a machine with complex data types and addressing modes. For such machines or others with a longer pipeline the length of the control latency may grow correspondingly. This leads to poor performance by using the delayed branch approach, because filling useful instructions in the later delay slots becomes extremely difficult.

Branch prediction

A pipeline with **branch prediction** guesses the outcome of a branch decision before it is determined. The pipeline prefetches the instruction stream from the predicted sequence and eliminates, therefore, the control hazard provided the prediction is correct. Otherwise, the pipeline must throw away the prefetched instructions and fetches the destination instruction renewly. In general, the pipeline can predict the branch path either *statically* or *dynamically*. A static prediction is determined by the compiler at the compile time, whereas a dynamic prediction uses the information of executed instructions and determines the branch path at run time.

The simplest form of static prediction assumes that all branches are taken or never taken. The related work showed that more than 50% of all branches are taken in general purpose applications. This means also that a prediction with *taken* leads to a better performance than a prediction with *never taken*. A variation of static prediction is to take different prediction according to the types of branches being executed. The pipeline predicts the branch path according to the branch types which can be identified in the related OP-code, namely, the pipeline predicts some branches to be *taken* and other branches to be *never taken*. The

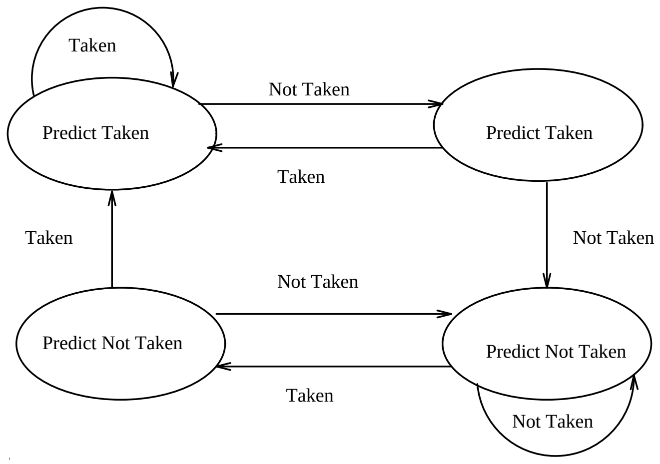


Figure 3.4: The two-bit prediction scheme

accuracy of predictions depends on the compiler's ability to prediction decision. The related work [Lil 88] reported that this approach reaches a prediction accuracy of more than 75%. Some further forms of static prediction are also possible. For example, in some processors a bit *likely/not-likely* is defined in the branch instruction format, the compiler sets all such bits at compile time using some information extracted from the program structure, for example, a branch for loop control should be set as *likely* etc.. During execution the hardware determines the branch path according to this bit. McFarling and Hennessy reported that the ratio of correct prediction using their algorithm reached up to 85% [McF 86].

Dynamic prediction, however, tries to determine the branch path using some history information of instructions having been executed at run time. Such history information is normally registered in form of a table in which each branch instruction is mapped to an entry. An entry in the table will be modified when the related branch instruction is executed. Lee and Smith evaluated several different dynamic predictions with stored history bits[SmL 84]. A well known two-bit-scheme is shown in Figure 3.4[McF 86], which shows the finite state machine for updating the table entry for a branch. As seen, a prediction must miss twice before it is changed. The average accuracy of prediction with this scheme reaches to about 85%.

Regardless of how the prediction is determined, the penalties for the branch prediction are given in table 3.2: There are four different combinations of predicted and actual branches; the penalty a is often equal to null, $b = c = L_c$. d is often not equal to zero, because the destination address is produced at the execution at a later stage of the pipeline. Even if the destination address is contained in the instruction, some unused cycles may arise until the destination instruction is fetched from the memory. In [Hen 90] the penalty a is equal to 1, b = c = 3, and d = 2.

In these cases the performance equation for the static prediction with *never taken* can be rewritten as follows:

Prediction	**Actual** Not taken	taken
Not taken	a	b
taken	c	d

Table 3.2. Control latencies for the branch prediction

$$CPI = 1 + bp_b p_v + p_d L_d \tag{3.12}$$

and for the static prediction with *taken*:

$$CPI = 1 + cp_b p_v + dp_b(1 - p_v) + p_d L_d \tag{3.13}$$

where p_v is the probability of a false prediction.

Example

Let the static prediction be *taken* and the accuracy of prediction be 85% ($p_v = 0.15$) (for example, with McFarling and Hennessy's algorithm for determining the bit *likely/not-likely*), $c = b = L_c = 3$, and $d = 2$. Using the same data as in table 3.1, we get $CPI = 1.367$, so that $M_f = 0.73$.

Note that the above example shows only the performance of the static prediction. For dynamic prediction it is difficult to give an exact prediction accuracy, because the probability that a prediction is false depends mainly on the current state as shown in the state diagram of Figure 3.4. This means that the probability p_v is a function of four states. To simplify the analysis some further probabilities are introduced as follows:

- p_{vn} is the probability of a false prediction by using *predict not taken*.
- p_{vt} is the probability of a false prediction by using *predict taken*.

thus $p_v = p_{vn} + p_{vt}$.

Correspondingly, two other probabilities can be defined:

- p_{cn} is the probability of a correct prediction by using *predict not taken*.

- p_{ct} is the probability of a correct prediction by using *predict taken.*

thus $p_v = 1 - p_{cn} + p_{ct}$.

Therefore, the performance equation of the dynamic prediction can be given as follows:

$$CPI = 1 + p_b(ap_{cn} + bp_{vn} + cp_{vt} + dp_{ct}) + p_d L_d \tag{3.14}$$

Assume $c = b = L_c$ and $p_{cn} = p_{ct}$, the performance expression above can be rewritten:

$$CPI = 1 + \frac{p_b}{2}((2L_c - d)p_v + d) + p_d L_d \tag{3.15}$$

Example

Let the accuracy of the dynamic prediction be 90%, $L_c = 3$ and $d = 2$. Using the same data as for the static prediction, we get $CPI = 1.22$, so that $M_f = 0.82$.

Branch target buffer

As mentioned above, the control latency d is normally not equal to zero, because the calculation of the destination address and the fetch of the destination instruction lead to some unused processor cycles. The approach called **Branch target buffer** (BTB) can eliminate this delay by employing a special cache in which an entry consists of the address of a branch instruction which has been executed at least one time and its target instruction itself, together with some prediction bits. While decoding a branch instruction BTB is associatively searched with its address. If the address is in the BTB and the prediction is correct, then the destination (target) instruction will be fetched for execution from the BTB, otherwise the old target instruction must be thrown away from the BTB, a new target instruction be loaded into the associated entry, and the prediction bits be modified.

So it is easy to get the performance form for the BTB approach as follows

$$CPI = 1 + p_b p_v (L_c + p_m C) + p_d L_d \tag{3.16}$$

where p_m is the miss rate of the BTB cache, and C the time for a cache miss.

Example

Let the probability p_v be 0.15, p_m 0.20 and C 2 (these values are extracted from the related work in [Hen 90][McF 86]). With the same data for the prediction approach we have $CPI = 1.12$, so that $M_f = 0.89$.

Mechanisms	CPI	M_f
Basic pipeline	2.68	0.37
Forwarding	1.64	0.61
Forwarding + delayed load	1.57	0.64
Delayed branch (basic)	1.57	0.64
Delayed branch (improved)	1.13	0.88
Branch prediction (static)	1.36	0.73
Branch prediction (dynamic)	1.22	0.82
Branch target buffer	1.12	0.89

Table 3.3: A comparison of different pipeline mechanisms

3.3.3 Summary and Remark

A summary of the performance for the different methods discussed above is listed in table 3.3. The pipeline assumed here is based on the DLX pipeline.

This table gives, however, no obvious detail of hardware costs in implementing those mechanisms, which actually influence the choice of methods. In an implementation we must consider the impact of clock cycle and the corresponding hardware costs, because the actual performance of a pipelined processor is influenced by both factors, as mentioned in the beginning of this section. The *delayed branch* is a pure software technique and requires only little hardware support (an additional program counter for the case of interrupt [Hen 90]). The delayed branch often reaches a satisfying performance for a short pipeline, whereas for a long pipeline other methods are necessary because of the difficulty in finding useful instructions for delayed slots. On the contrary, the BTB does not require any software support, which simplifies the design of a compiler. The hardware overhead of the BTB is, however, very notable and prevents this technique from practical application. For example, a BTB with 256 entries may be so large as the rest of the processor in a single-chip implementation [McF 86]. The branch prediction has been used in many modern microprocessors. For examples in the newest PA-RISC processor (a superscalar processor) the prediction strategy is: a *forward* branch is predicted as *not taken*, whereas a *backward* branch is assumed to be *taken* [Del 92]. Many other variations of branch mechanisms have also been proposed in recent years such as *branch bypass and multiple prefetch, branch folding, early branch resolution* etc. [Lil 88].

The discussion of this section has shown us two important facts:

- all methods only have the goal to reduce the performance loss due to pipeline hazards, but can not avoid or eliminate pipeline hazards completely, because the conventional pipeline improves the performance by exploiting limited parallelism from a *single* instruction stream.
- the improvements of performance require either complicated software tools such as a *clever* compiler or high cost hardware mechanisms.

With increasing requirements on processor performance it is desired to build a faster pipeline, which means often also to build a deeper pipeline. This leads, however, to more serious problems with pipeline hazards. To reduce the effects of such hazards, more software or hardware mechanisms must be introduced. Are there any alternatives to the conventional pipelined architectures? The answer is *yes*. We will answer this question in the following sections.

3.4 Extensions to the Conventional Pipeline – VLIW, Superscalar and Superpipeline

Before we deal with the question of some alternative to the conventional pipelined architectures, we will first examine the three most important developments based on the conventional pipeline concepts. The purpose of this section is to understand how much these extensions can influence the performance of processors and what their restrictions are.

3.4.1 VLIW - Very Long Instruction Word

VLIW takes advantage of the instruction parallelism to reduce the number of instructions, i.e. the first factor. In a VLIW processor, multiple concurrent operations are specified in a very long instruction word (e.g. several hundred bit wide), thus multiple operations can be performed by issuing only one instruction as shown in Figure 3.5.

Because a single VLIW instruction can specify multiple operations, VLIW processors reduce the number of instructions in comparison to pipelined processors. However, to obtain this performance advantage, the operations specified in a VLIW instruction must be independent of one another. This means that the performance of a VLIW processor is strongly dependent on the associcated software to find such independent operations for each VLIW instruction. The more concurrent operations can be packed into every VLIW word on average, the better the performance of the processor.

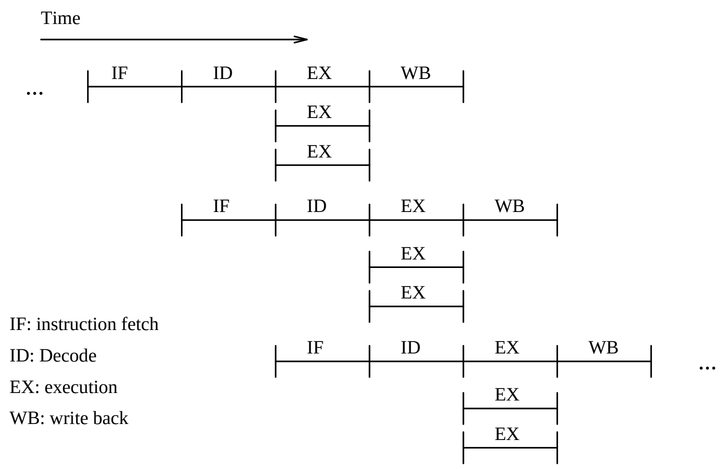


Figure 3.5. Instruction execution in a VLIW processor

To accomplish this, a software technique called *compaction* is often used [Joh 91]. The *compaction* software searches the concurrent operations from the instruction stream and packs them together into VLIW words. During compaction, *no-ops* (null operation fields) must be inserted into operation fields if these operations can not be used. In the AP-120B[2] of Floating Point System [Cha 81], one of the earliest VLIW machines, a well known technique called *software pipelining* was first introduced in writing the routines for its mathematical library with 64 bit microprogram words. Software pipelining reorganizes loops such that each iteration in the software pipelined code is made from instruction sequences chosen from different iterations in the original code segment, so that a new loop interleaves instructions from different iterations without unrolling the loop. *Loop unrolling*[3] is another well known technique used for packing VLIW words. Loop unrolling replicates a loop multiple times. The unrolled loop then contains more possible concurrent instructions which can be renewly scheduled to pack possible concurrent operations into VLIW words. A further method is called *trace-scheduling* which was originally developed for VLIW processors. Trace scheduling consists of two separate processes called *trace selection* and *trace compaction* for finding the most likely instruction sequence of operations to put together and for compacting these instructions into as few VLIW word instructions as possible.

The main advantage of VLIW processors over pipelined processors is that the number of instruction can be dramatically reduced if application programs contain a large amount of instruction parallelism just as in many scientific computations. As general purpose machines, however, VLIW processors have their inherent limitations:

[2] Differently from other VLIW processors developed later, AP-120B is not a complex processor and was designed only as an attached array processor to certain general purpose machines such as PDP-11 and accepts the array processing task from the general purpose machine.

[3] Loop unrolling is also used in pipelined processors to enlarge *basic blocks*, in which more instruction parallelism can be exploited.

- limited parallelism - the parallelism exploited from a single instruction stream is limited. For many general purpose applications there may be not enough parallelism to fill operation fields in VLIW words. Provided functional units, memory units, branch units, floating point units etc. will be pipelined, a much larger number of operations can be executed in parallel. For example, a VLIW processor with 8 functional units may require 20-30 operations in order to keep all functional units busy. The developed VLIW processors are mainly suitable for applications with a large amount of parallelism, such as AP-120B [Cha 81] developed by Floating Point System and iWarp by Intel and Carnegie-Mellon [Coh 89].
- hardware cost - the required hardware cost lies not only in duplication of functional units such as floating point units, but also in a large increase in the memory and register bandwidth which provides a large amount of data flow to the functional units. A typical configuration of memory units contains an integer register file with 7 read ports and 3 write ports, a floating point register file with 5 read ports and 3 write ports and a main memory with 2 ports [Hen 90].
- code size explosion - the code size of a compacted code may increase rapidly. Two different elements will contribute to rapid code increase: first, generating enough instruction parallelism requires often ambitiously unrolling loops, which increases the code size; second, many no-ops must be filled in operation fields if no suitable operation can be found, which leads to the waste of instruction encoding.
- software compatibility - the most important limitation is that VLIW processors are not software compatible with any general purpose processors, because a VLIW program is specially reorganized for a particular VLIW processor.

It should be noted that the problem with pipeline hazards is not viewed as a critical issue in VLIW processors, because all pipeline hazards must have been statically resolved by the scheduling software at the cost of more no-ops inserted in operation fields.

3.4.2 Superscalar Processor

While VLIW processors improve their performance through exploiting instruction parallelism statically, **Superscalar** processors select concurrent instructions to execute dynamically at run time, which leads to execution of more than one instruction per clock cycle. Therefore, superscalar processors improve their performance by reducing CPI further. The execution principle of a superscalar processor is shown in Figure 3.6. The broken line depicts a no-op.

By issuing more than one instruction to execute, a superscalar processor may reduce CPI below 1. However, the performance of a superscalar processor depends not only on instruction

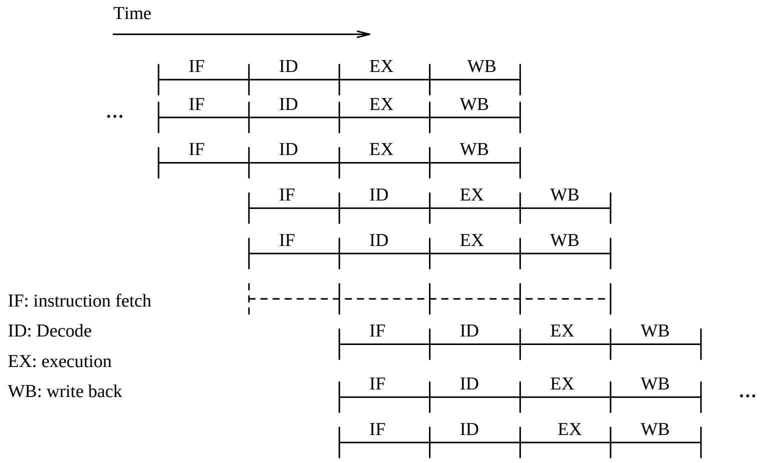


Figure 3.6. Instruction execution in a superscalar processor

parallelism contained in application programs, but also on hardware supports for exploiting available parallelism dynamically and for detecting and resolving the more serious problem with pipeline hazards. Further, a superscalar processor may have several additional issues: *out-of-order-issue* of instructions, *register renaming, precise or imprecise interrupt, multiple instruction issue* etc. [Joh 91]. A superscalar processor must support enough machine parallelism to execute multiple instructions at any time, duplicate resources to avoid structure hazards, support *out-of-order-issue* to exploit more concurrent instructions, and provide a *register renaming* mechanism to detect and resolve *anti-* and *output* data dependencies. Since more than one instruction may be fetched to execute at any time, the *run length* of basic blocks is decreased, which increases, however, the problem with control hazards. The techniques developed for control hazards in conventional pipelined processors may not be applied directly, because the superscalar processor can decode more than one branch instruction, may have to retain a branch instruction several cycles before it is executed, and may also have several unexecuted branches pending at any given time. Johnson reported in his book that there are average 2.4 branch instructions pending in a superscalar processor with four instruction decoders [Joh 91].

Because of dynamic issuing of instructions, more hardware and software support are required in comparison to VLIW processors. Other features are included as follows:

- Decoding of instructions becomes more difficult, because the decode unit must check all possible conflicts and dependencies dynamically at run time, while any VLIW instruction has a fixed format, and software is responsible for detecting and resolving all conflicts and hazards statically.

- If the average instruction parallelism in a program is less than one supported by VLIW processors, then superscalar processors have better code density, because superscalar processors contain only useful instruction bits while a large amount of no-ops must be filled in VLIW instructions.

- Superscalar processors can be object-code compatible with many general purpose processors, while VLIW processors possess their own quite different instruction format, which is impossible to be compatible with any general purpose processors.

Because of hardware and software complexity and of limited instruction parallelism in general purpose applications, most of the newest superscalar processors such DEC's Alpha, Motorola's M88110, Hewlett-Package's superscalar PA-RISC, Intel's i860 etc, therefore support issuing a maximum of two instructions per clock cycle. To issue more instructions, more hardware mechanisms such as instruction window, result window and their associated control logic are required [Joh 91].

3.4.3 Superpipelined Processors

A **superpipelined** processor tries to increase the clock frequency by splitting the major stages of a pipelined processor further into substages. In a superpipelined processor one instruction can be fetched to execute on every subclock cycle, therefore, a superpipelined processor improves its performance by reducing the time t_p, i.e. the last factor influencing the execution time of a program. The basic operation principle of a superpipelined processor is shown in Figure 3.7. As before, dashed lines represent no-ops.

The basic idea of superpipelining has been used in many vector processors. In their work [Jou 89] Jouppi and Wall compared the features of superscalar and superpipelined processors and declared that superscalar and superpipelined processors with the same *degree* [4] have basically the same performance. Superpipelined processors may reach the same performance with less hardware cost.

The performance of a superpipelined processor may be mainly influenced by three factors: more pipeline hazards because of its longer pipeline, latch overhead and clock skew[5]. The last two factors determine the clock cycle time as follows:

$$t_p = max(t_i) + t_s + w \tag{3.17}$$

where t_i is the delayed time of the substage i, w the time for a latch to accept a result and t_s the clock skew. It should be noted that in a superpipelined processor with high clock frequency,

[4]In superscalar processors, *degree* is the maximum number of instructions issued per clock cycle, while in superpipelined processors *degree* is defined to be the number of substages.

[5]A clock skew is defined to be the relative timing of the physical clock signals that control various portions of the processor.

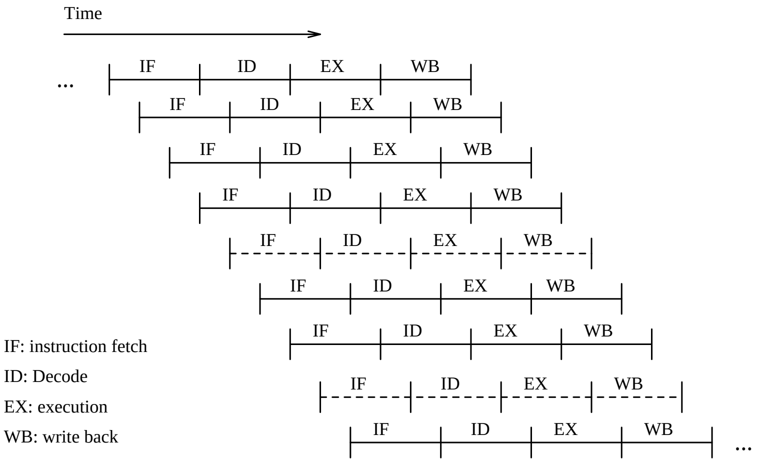


Figure 3.7. Instruction execution in a superpipelined processor

the effect of clock skew is very notable. For example, a 100 MHz superpipelined processor with the degree of 2, a skew of 2 ns represents 20% of the minor cycle time. Superpipelining is, therefore, an appropriate technique when the cost of resource duplication is high and the ability to control skew is good. This means that this techniques is suitable for processors implemented in very-high-speed technologies.

One of the newest superpipelined processors is MIPS's R4000 which consists of 8 stages and was developed with the following considerations [Mil 92]:

- less hardware overhead is required in comparison to a superscalar implementation with the same degree, because no duplication of internal functional units is necessary.
- the control logic is much simpler, because no logic for issuing multiple instructions and detecting possible dependencies dynamically is required.
- complete binary compatible with available software without any recompilation.
- faster clock cycle and shorter design and test time.

Note that superpipelined processors inherit all problems from the pipelined processor. Moreover, the operation latency grows correspondingly because of their longer pipeline. For example, the length of control hazards increases to three in R4000 instead of one in R3000, which leads to 20 - 30% performance loss due to no-ops or pipeline stalls [Mil 92].

3.4.4 Summary and Remarks

As described, VLIW, superscalar and superpipelining are three extensions to conventional pipelining to reduce three factors influencing the execution time of a program, respectively. A hybrid form of the above methods is possible like some vector processors which can be viewed as a hybrid of superscalar and VLIW, because such processors contain multiple pipelined functional units and execute multiple operations and instructions.

The potential performance advantage of these techniques can only be achieved by exploiting instruction parallelism in a single instruction stream. There are three aspects which must be carefully considered in using these techniques:

- instruction scheduling,
- pipeline hazards and
- memory throughput.

An instruction scheduling mechanism should issue as many executable instructions as possible under the limitation of the machine parallelism and must not change dependencies of the original instruction sequence. To accomplish this, both hardware and software support are necessary. Hardware mechanisms such as *grouping logic* in SuperSparc [Bla 92] as well as *instruction sequencer* in M88110 [Die 92], fetch independent instructions for simultaneous execution. Software support, especially an optimizing compiler, is needed to exploit possible instruction parallelism, which can be done at two levels: *within* basic blocks and *across* basic blocks. At the first level, all dependencies between instructions within a basic block must be represented by, for example, a *dependency graph*. A software scheduler assigns instructions to functional units according to certain algorithm such as *list scheduling*. At a higher level, the scheduler should minimize effects of branches across basic blocks, so that the parallelism between basic blocks can be exploited. For this purpose there are three software techniques: trace scheduling, loop unrolling and software pipelining as mentioned in the last section. Because of requirement of more information throughput, memory throughput has become another critical issue which may lead to high hardware cost and therefore should be carefully balanced with other factors.

3.4.5 Restrictions and Problems

As seen, pipeline hazards are the most important factor which influences directly the performance of a pipelined processor or its extended processors. To reduce the effects of pipeline hazards further, both *clever* software and *more cost effective* hardware mechanisms are needed. One can see this situation in some of the most recent microprocessors such as M88110, SuperSparc, Alpha, PA-RISC etc.. For example, M88110 superscalar processor contains ten parallel functional units and can issue, however, maximally up to two instructions to these functional units. One obvious question is why the utilization of hardware resources is so low? Is such *generous* solution to structure hazards cost effective? The same question can also go to the mechanisms used to detect and resolve the data and control hazards: *branch target instruction cache, reservation station, history buffer* etc.. So it is not surprising that these new processors have been implemented in more than 1 million transistors[6]. Is this a cost-effective way to exploit increasing silicon resource?

In almost all newest microprocessors, instructions are issued for execution in exact program order[7]. Different operations may have different execution time, which leads to *out-of-order-completion* and can result then in pipeline-bubbles. The dependencies due to out-of-order-completion can be resolved by using *register renaming* in theory, but at the cost of an expensive and complicated hardware mechanism. Furthermore, the increased instruction and data throughput may cause more cache misses, which lead to idle time of the processor while new data or instructions are loaded from memory.

To further improve the performance of a computer system, it is strongly desired to build multiprocessor systems consisting of multiple single processors. If we use the above processors which operate actually based on the *von Neumann* principle as *processing element*, such multiprocessors will suffer from two fundamental issues mentioned by Arvind [Arv 87], i.e. ability to tolerate memory latency and ability to synchronize shared data without constraining parallelism.

We now come back to the question of the last section. Why are we falling into a dilemma in which we have to pay much but achieve only a little in improving processor performance? The most important lesson we have learned in the discussion above is that we have been trying to solve pipeline hazards on the basis of exploiting instruction parallelism from a single instruction stream. It has, however, been proved that the parallelism from a single instruction stream is very limited, and all methods described can only reduce the effects of pipeline hazards, sometimes at a high hardware cost, but not avoid them completely. A further question is why we do not accept the fact that pipeline hazards are inevitable and look for some means of masking them instead of reducing their effects with complicated software or hardware mechanisms. An alternative to conventional pipelined architectures is to maintain multiple instruction streams instead of only one instruction stream in the processors and switch among them when some event (e.g. pipeline hazard) occurs. This is the technique called **multithreading**, the main theme in the following sections.

[6] M88110 contains 1.5 million transistors, Alpha 21064 1.7 millions and SuperSparc 3.1 millions.

[7] To the author's knowledge, even the newest superscalar processors support no *out-of-order-issue*, which was discussed in Johnson's book[Joh 91].

3.5 Multithreading - An Alternative

Unfortunately, there is no generally accepted definition of the concept of multithreading. In this section we will explain its fundamental features and then try to outline a clear concept of multithreading. After that, some known multithreaded processors will be sketched briefly.

3.5.1 The Concept of Multithreading

From Multiprogramming to Micromultiprogramming

Multiprogramming is a technique which has been used for optimal utilization of the expensive hardware resource in many high performance computers for a long time. To get reasonable utilization, the time slice allocated to each process must be large in comparison to the overhead of context switching. Multiprogramming operates therefore at the level of the operating system. If we use the same technique at a lower level such as instruction level, some additional mechanisms must be provided to switch the context quickly. Such multiprogramming at the instruction level is called **micromultiprogramming** [Top 88]. With micromultiprogramming, multiple instruction streams are interleaved in execution in the instruction pipeline, so that each process has one, and only one, instruction in a partial state of execution at any instant, which contrasts sharply with conventional instruction pipelines where as many instructions as possible are kept active for a single instruction stream. One obvious consequence of using multiprogramming is that all pipeline hazards disappear now, because instructions overlapped in execution in the pipeline come from different instruction streams and are therefore independent of each another. It is also clear that a fundamental requirement on such processors is to provide an efficient context switching mechanism which should cause no or little time overhead. A comparison of micromultiprogramming to conventional pipelining is shown in Figure 3.8.

Thread

On basis of micromultiprogramming we define a *thread* as follows:

> **Definition 3.5:** A **thread** is a tuple $< T, T_e >$, where T is a sequential instruction stream and T_e all its volatile information called **execution environment**. □

This definition includes not only an instruction stream which is executed sequentially, but also its execution environment. In practice, the identifier of a thread and its environment are represented by a structure called **thread descriptor** which refers to the instruction stream

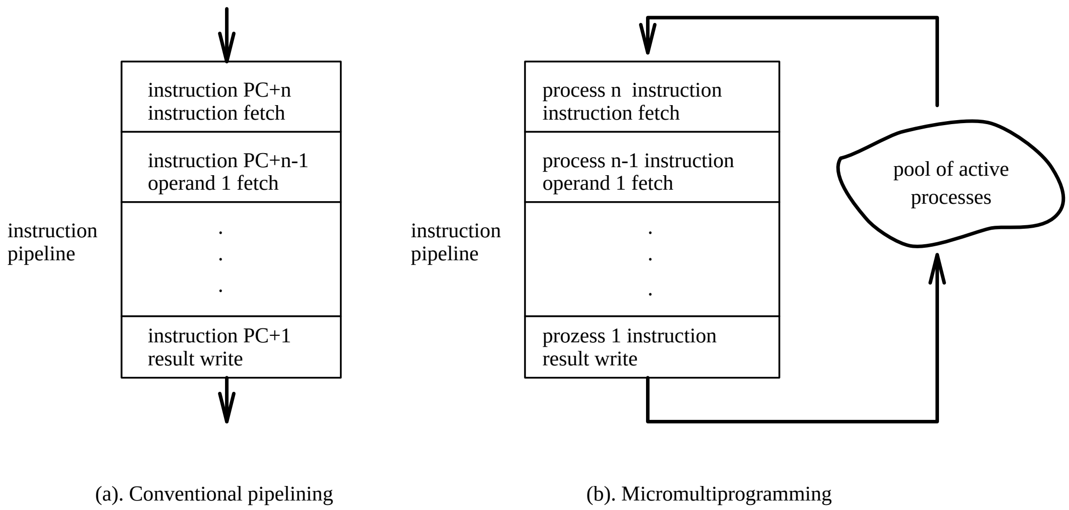


Figure 3.8. Pipelining vs. micromultiprogramming

and the associated environment. An actual execution environment may be divided into two different parts: *dynamic environment* and *static environment*. The dynamic environment contains the information used or modified in execution of each instruction, such as the program counter, and is therefore stored in the processor; the static is the rest environment consisting of, for example, a register set, a segment of memory (frame) etc.. The static environment is often much larger and must be referenced by some *pointer structure*.

It should be noted that the meaning of *thread* may differ in the different cases. In multiprogramming environments, a thread corresponds to an independent instruction stream (also called process) which consists of a complete user program or a system process. The corresponding thread descriptor in such systems is often called *system status word* or *task status word*. In multiprocessor environments a program is divided into a set of small tasks which are executed in parallel and communicate with each other. Each task corresponds to a thread which can be as small as a single instruction or as large as a prodecure. A thread is also known as *scheduling quanta* in Iannucci's hybrid machine [Ian 90], *task* in MASA [Hal 88], *process* in PUMA [VdH 86]; and the thread descriptor as *continuation*, *register specifier* and *process descriptor*, respectively.

Multithreading

Based on the definition 3.5 we define the concept of *multithreading* as follows:

> **Definition 3.6: Multithreading** is the *interleaved* execution of multiple threads and represented by a tuple $< T_s, S_s >$, where T_s is a set of threads and S_s the scheduling strategy of the controlling context switching among threads.
>
> □

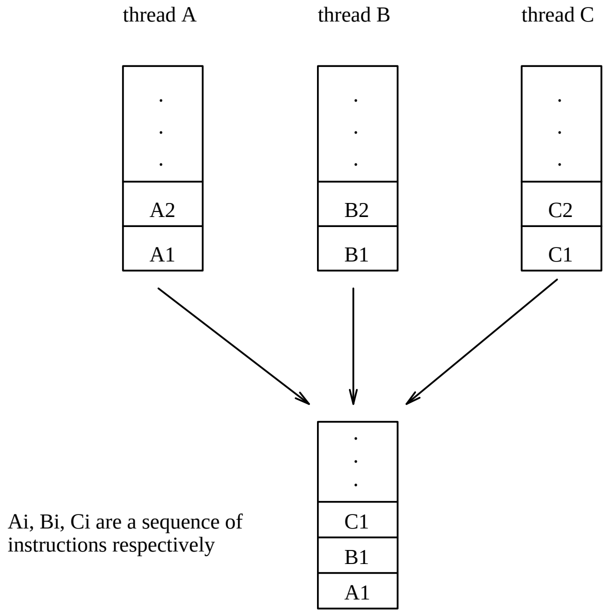


Figure 3.9. Multithreaded execution

The scheduling strategy S_s in the definition determines when and how a thread is switched off from its execution. The scheduling strategy is often determined by the occurence of some *event* such as pipeline hazards, cache miss etc. We will discuss several different scheduling strategies in the next chapter. The simplest form of the scheduling strategy is to switch thread on every clock cycle. This means a thread can only be switched in to execute after one instruction from the same thread has left the instruction pipeline. Thus each stage contains an instruction from a different thread (or pipeline bubble), i.e. micromultiprogramming. We call a pipeline augmented with multithreading mechanisms a **multithreaded pipeline**. The basic principle of a multithreaded execution is shown in Figure 3.9. The concept of multithreading is similar to other concepts known as *instruction interleaving* [Sti 91b], *virtual-processing* [Fly 72], *multiple-instruction-stream* [Kam 79], *contextflow* [Top 88], *circulating-contexts* [Sta 86b], etc.

3.5.2 Fundamental Features of Multithreaded Processors

The original motivation in developing multithreaded processors has been **efficiency** in using the hardware resource, just as the one for multiprogramming. T.C. Chen first introduced the concept of micromultiprogramming and described a mechanism for sharing the hardware resources of a parallel or pipelined system[Che 90]. Another well known form of micromultiprogramming was proposed by Flynn [Fly 72], in which highly pipelined functional units are shared between 32 *skeleton* processors. Further work was done by Kaminsky and Davidson [Kam 79] who proposed to use micromultiprogramming in the design of single chip processors in order to utilize the chip resource efficiently, especially, the efficient use of off-chip connections. Recently, Park and his colleagues [Par 91] studied the performance advantages of multithreading in designing modern microprocessors and observed: "*in a modern parallel environment, multithreaded processors provide a cost-effective way to exploit increasing silicon resource to achieve higher performance.*"

Multithreading is also a **solution to pipeline hazards** in a pipelined processor. As mentioned, using the scheduling strategy *switch on every cycle* instructions in all pipeline stages come from different threads, therefore no pipeline hazards may arise, because the instructions overlapped in execution are independent. If enough threads are available, then the multithreaded pipeline achieves its maximum performance, i.e. one instruction per cycle. Because no hardware mechanism is required for detecting and resolving pipeline hazards, this allows to build a faster pipeline. We see here that an obvious problem with the multithreaded pipeline based on *switch on every cycle* is its poor single thread performance, because each thread uses only a part of the pipeline resource during execution. If not enough threads are available, no-ops must be inserted in the pipeline stages. There are several alternatives to this scheduling strategy, which are described in detail in the next chapter.

In a multiprocessor environment, multithreading is a technique to **hide long memory latency**. As mentioned, in multiprocessing parallel threads must communicate and synchronize each other in order to obey the desirable computation order. Such communication or synchronization is realized either through some *shared variables* in a shared memory system or through *message passing* in a distributed system. The delay or latency between issuing a request and receiving a related response is often very long and unpredictable due to network distance and conflicts. Multithreading can hide such long latency as follows: when an operation causing long latency is encountered, the processor suspends the running thread and switches to another active thread. After receiving the response from either a remote memory or a processor, the processor makes the suspended thread executable again. If there are enough parallel threads in the processor, the memory latency can be masked completely.

A further feature of multithreading is to **support asynchronous MIMD (Multiple Instruction Multiple Data) computation** directly on a single processor. If a multithreaded processor is augmented with mechanisms for synchronization, this multithreaded processor is itself a multiprocessing processor. The main advantage is that such processors can exploit both intra-thread parallelism just like conventional pipelined processors as well as inter-thread parallelism, i.e. the parallelism from multiple instruction streams. Therefore, a multithreaded uniprocessor can exploit more parallelism than any other processors such as VLIW, superscalar, and superpipelined processors. The only limitation of extractable parallelism on a multithreaded processor is its machine parallelism. Moreover, the data flow activities have strongly influenced the research in multithreaded architectures because of their ability to tolerate long latency, their cost effective implementation and their software compatibility with existing systems.

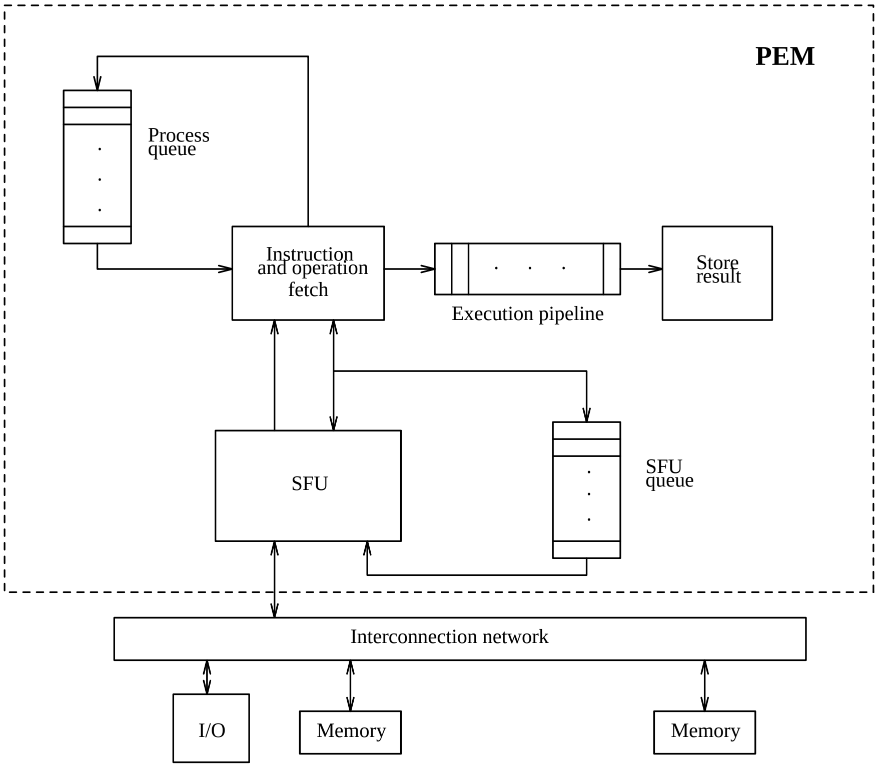


Figure 3.10. The architecture of HEP

3.6 Some Examples of Multithreaded Processors

Several multithreaded processor designs have been proposed or implemented in recent years. The common properties of these processors are their ability to improve the hardware resource utilization and to avoid the two fundamental problems mentioned by Arvind. In this section, we will outline some important multithreaded processors.

3.6.1 Heterogeneous Element Processor (HEP)

One of the best examples of multithreaded processors to date is the Denelcor Heterogeneous Element Processor, which was first described by B. Smith in 1978 [Smi 78] and was produced several years later. The architecture of HEP is illustrated in Figure 3.10. HEP is a shared memory multiprocessor consisting of up to 16 *process execution modules* (PEMs). Each PEM is a multithreaded processor, in which the scheduling of instructions is based on the strategy *switch on every cycle*. The processor module supports up to 64 processes (threads) and contains an execution pipeline with 8 stages. Furthermore, a large general purpose register file (2048 words) and a large constant register file (4096 words) are used to maintain multiple processes in the processor in order to support quick context switching on every clock cycle.

Two novel features of HEP are its ability to tolerate the memory latency through multithreading and its low level synchronization mechanisms in the form of *presence-bits* in registers and the main memory. All accesses to the memory will not be issued to the execution pipeline but are enqueued into the *memory reference pipeline* in the SFU (Storage Function Unit), which then routes them further to the memory. Thus, these operations do not block the execution of instructions from other processes. After a successful access to the memory, i.e. after receiving the response from the memory, the process waiting in the SFU buffer is then activated and inserted into the execution pipeline again. If an access has failed, for example, because certain data is not available in the memory, this access will be routed to the memory again in the next turn. The presence-bits allow synchronization at the data level. This synchronization mechanism or its variations such as the I-structure proposed by Arvind have been widely used in many data flow machines or other multiprocessors.

Two critical points of HEP are the *limit of 64 processes* in each processor, which has been viewed as a serious impediment to the software development on HEP, and its *busy-waiting* strategy of handling failed remote accesses, which can consume much hardware resource, especially network resource, if there is a large amount of such failed accesses during execution. Further limitation is that each process can only issue one remote access to the memory, which may limit possible parallelism in programs. Because of the scheduling strategy of switching on every clock, HEP suffers also from a poor single thread performance.

3.6.2 Parallel RISC (P-RISC)

P-RISC was proposed by Nikhil and Arvind as an alternative to dataflow processors. As mentioned in their paper, the goal of P-RISC is to synthesize the features of von Neumann and dataflow machines, so that the new processor "*can be viewed as a dataflow machine that can achieve complete software compatibility with conventional von Neumann machines*" [Nik 89].

P-RISC is built on the basis of a RISC machine by introducing additional hardware supports for multithreading and by applying the concept of *split-phase transaction* to every remote access. The schematic architecture of P-RISC is illustrated in Figure 3.11, where *tokens* correspond to thread descriptors, and each thread is executed in its own *frame* - a segment in the local memory.

Two simple instructions, **Fork IPt** and **Join x**, are introduced for initializing a new thread and for synchronizing parallel threads respectively. These are two simple instructions – not operating system calls – that are executed entirely within the normal processor pipeline. One of the novel properties of P-RISC is its synchronization instruction **Join x** which realizes synchronization between parallel threads by modifying an entry count in their frame. Thus the thread synchronization is performed explicitly without any presence-bits in registers or

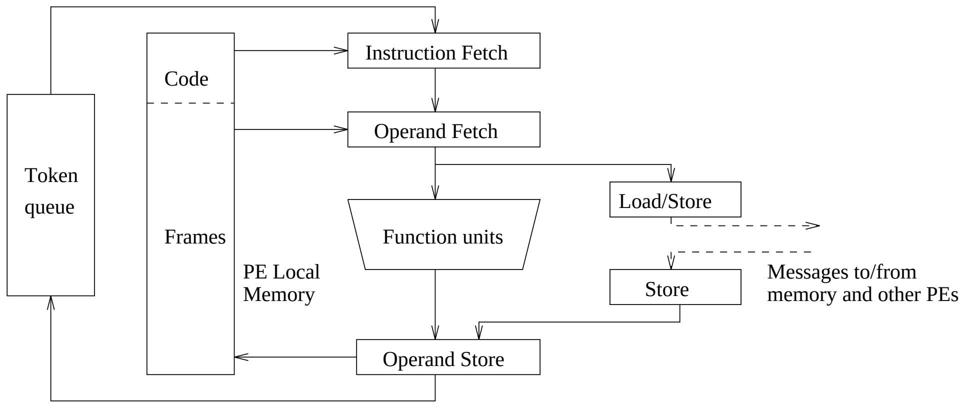


Figure 3.11. The architecture of P-RISC

in frames. Another novel feature is to the application of *split-phase transaction* to remote Load/Store instructions: remote load/store instructions are split into two *phases* which seperately initiate the request and later synchronize prior to using the result.

Many P-RISC processors may be connected with a global I-structure memory to build a multiprocessor system, which can execute both dataflow programs and conventional imperative programs. Such a multiprocessor system should have a better performance/cost ratio than that of dataflow machines, because the processor element is based on a familiar von-Neumann processor, and the realization complexity of dataflow machines is thus avoided. P-RISC was only proposed as a new architecture model and not built. Its successor *T is being developed by Arvind's group at MIT [Nik 92].

3.6.3 APRIL

APRIL is a further multithreaded processor developed as processing element in MIT's large scale multiprocessor system ALEWIFE [Aga 90]. APRIL was designed with the goal that processors in large scale multiprocessors must be able to tolerate large communication and synchronization latencies and contain an efficient context switching strategy. APRIL supports directly the programming language Mul-T, an extended version of Scheme.

The design of APRIL is strongly influenced by Halstead's multithreaded processor MASA [Hal 88]. To improve the single thread performance, APRIL rejected the *switching on every cycle* strategy and introduced a so called *coarse-grain multithreading* mechanism: the processor continues executing a single thread until a memory operation involving a remote request (or an unsuccessful synchronization attempt) is encountered. The processor switches to another thread while the request is serviced. Unlike other multithreaded processors such as HEP or P-RISC, the processor must spend several additional cycles to switch a thread, because the processor uses a trap handler to realize context switching. Some further features of APRIL

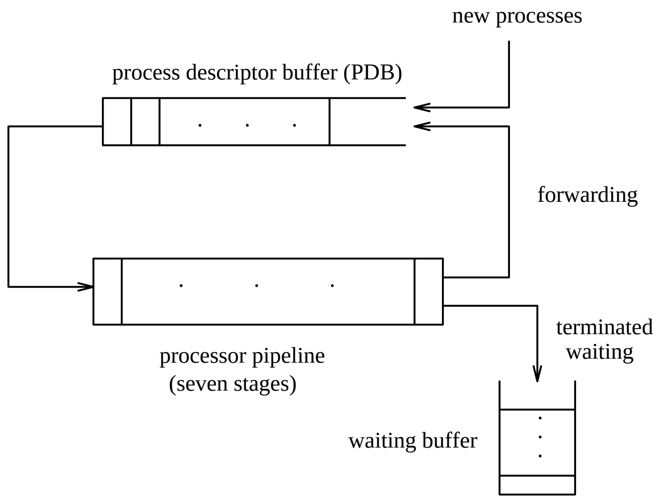


Figure 3.12. The logic architecture of PUMA

include support of the *future* concept - an mechanism to initiate parallel tasks in Multilisp [Hal 88], and *lazy task creation* - a method used to optimize the execution time by avoiding generation of many unnecessary parallel threads [Aga 90].

An implementation based on a Sparc processor is available. Eight register windows of the Sparc processor are reorganized into four task frames, each consists of an user window and a trap window. The overhead for a context switching is 11 processor cycles. Because the processor supports coarse multithreading, a few threads are required to hide the memory latency (4 in this implementation). It is not clear how much the overhead for context switching will influence the performance of the whole system. Loading and unloading threads are expensive operations, there is yet no special hardware mechanism for this purpose. The experimental results reported in [Aga 90] were extracted only from a simulation environment which excluded many issues.

3.6.4 Buffer Machine (PUMA)

PUMA (Puffer Maschine in german) is another multithreaded processor which influences the author's work directly[VdH 86]. PUMA is basically an object-oriented and fault identifiable processor proposed by von der Heide, and was also implemented by a team at the *Forschungsinstitut für Funk und Mathematik* (FFM). PUMA was designed with the goal of a high reliability of hardware and software. Here we will only outline the multithreaded features of PUMA. The logic architecture of PUMA is shown in Figure 3.12.

A thread is called *process* in PUMA, which is identified by a *process descriptor* in which the necessary references to code and data as well as some bits for fault identification are included. The logic architecture of PUMA consists of 7 pipeline stages, together with a programmable buffer - process descriptor buffer (PDB) in which all active process descriptors are resident - building a ring of instruction execution. The process scheduling is based on *switching on every cycle*. The instruction execution begins with fetching a process descriptor from the PDB and ends with writing the process descriptor into the PDB again or into the waiting buffer if the process is ended or must wait for some synchronization conditions. All codes and data are organized in form of *objects* which can be accessed only through the corresponding *object descriptors*. Indeed, the PDB and the waiting buffer are objects and not built in special hardware. The PDB may be exchanged by one instruction with the effect of activating and deactivating groups of processes.

3.6.5 Summary and Remark

Multithreading as an alternative to conventional pipelined processors has gained great attention in recent years. Modern microprocessors such as superscalar and superpipelined processors achieve their high performance by introducing two important techniques at the architectural level: integrating multiple functional units on the chip and scheduling multiple instructions on every cycle. Two new side effects of these techniques are that more hardware and software mechanisms are required to reduce the effects of pipeline hazards and poor utilization of hardware resources due to limited parallelism extracted from a single instruction stream. Multithreading can mask all such problems by exploiting inter-thread parallelism. A multithreaded processor such as APRIL can also exploit intra-thread parallelism in order to improve single thread performance. Therefore, a multithreaded processor is expected to have a better performance/cost ratio than superscalar or superpipelind processors by exploiting all possible parallelism both at inter-thread and at intra-thread level.

Another important feature of multithreading is its ability to tolerate long and unpredicable latency in a multiprocessor environment. A multithreaded processor contains the basic properties of a von Neumann processor, at the same time it can easily be extended to execute dataflow programs just like P-RISC, without any complexity of conventional dataflow machines, and possesses the ability to handle the two fundamental issues of latency and synchronization. Multithreaded processors can also be viewed as a *tradeoff* or a *synthesis* of two different architectural models. Burton Smith, the chief architect of the Denelcor HEP expressed in his speech "The End of Architecture" [Smi 90] - *Many of us now agree that a hybrid von Neumann-data flow approach, with multiple instruction streams per processor, is preferable to pure data flow, but we disagree on the details of the stream architecture.*

Chapter 4

Performance Evaluation of Multithreaded Execution

While the multithreading technique is conceptually appealing, little work on actual performance evaluation of multithreaded execution has been reported. In order to understand the potential gains offered by this approach and its limitations, we will discuss the issues of actual performance of multithreaded execution. In this chapter, a classification of different multithreading models will be first outlined according to the strategies of thread switching (context switching) in order to extract the common properties of proposed multithreading models. After this, several basic parameters are extracted from those properties and used to express the performance of different multithreaded execution strategies. Subsequently, some simulation results based on one more realistic architecture will be presented. The analysis in this chapter influences directly the work presented in the next two chapters.

4.1 A Classification Scheme of Multithreading Models

All multithreaded processors maintain the state of several instruction streams (threads) and switch among them based on some event. Two important properties of multithreaded processors are, therefore, the associated cost caused by context switching, and the number of threads residing in the processor. These two properties are mainly influenced by *when* and *how often* thread switching is performed. Hence, it is meaningful to classify possible multithreading models according to their different thread switching strategies. Figure 4.1 shows such a classification based on this consideration.

4.1.1 Multithreaded Uniprocessors

1. Switch on every cycle (SOEC)

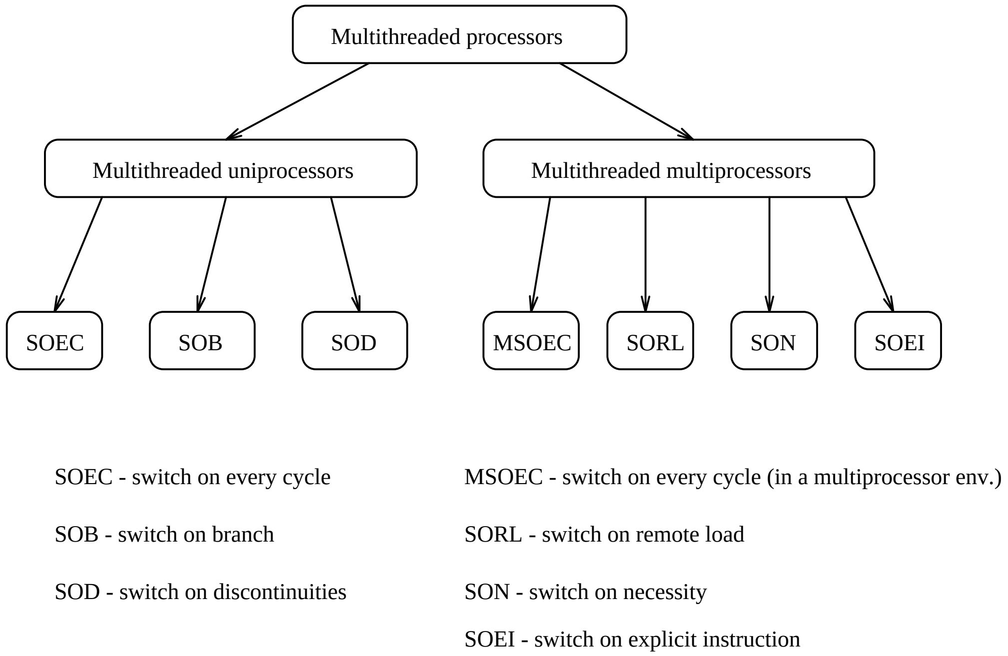


Figure 4.1. A classification of multithreading models

SOEC or *cycle-by-cycle (CBC)* is the oldest model of multithreaded execution (also known as micromultiprogramming) and was first proposed to improve the hardware utilization of some expensive hardware resources. Recently, SOEC was also used to mask the pipeline hazards in deeply pipelined processors [McC 91]. HEP might be the first processor which uses SOEC to tolerate the memory latency [Jor 85] – a problem which must be efficiently handled in building scalable multiprocessor systems. With SOEC the processor switches to another active thread on every clock cycle. At any time, each thread has one, and only one, instruction in a partial state of execution (i.e. in the instruction pipeline). All instructions which are executed in the pipeline belong to different threads and are therefore independent from each other. This allows to build a faster execution pipeline, because no special hardware for detecting and resolving pipeline hazards is required, which leads to simple logic between pipeline stages. The throughput of such a switching model would clearly be proportional to the number of concurrent threads, up to a maximum of the number of pipeline stages. Unfortunately, this model has a poor single thread performance, because each thread uses only $1/n$ part of the processor resource, where n is the number of the pipeline stages.

2. Switch on branch (SOB)

With this approach the processor switches to another active thread only when some branch operation is encountered. SOB can thus be viewed as an alternative to resolve the branch problem which belongs to the essential problems in pipelined processors. In comparison to SOEC, SOB has a better single thread performance, because the processor continues executing instructions from the same thread until the next branch instruction is encountered. The *switch* *distance* is, therefore, equal to the length of *basic blocks*. One obvious disadvantage of this model is that other kinds of pipeline hazards such as data hazards and structure hazards must also be handled. Some hardware mechanisms or pipeline bubbles are needed to resolve such hazards. This model is, hence, suitable for processors with a small number of pipeline stages, where the mentioned pipeline hazards arise relatively rare, or can be easily resolved. Iannucci's hybrid processor employs this scheme to avoid the branch problem [Ian 90].

3. Switch on discontinuities (SOD)

A direct extention to SOB is obvious, i.e. the processor switches to another active thread when any pipeline discontinuity (hazard) is detected. Differently from SOEC, the processor must contain some mechanisms to detect all possible pipeline hazards. The *Resource reservation table* as in M88110 can be used to detect all resource conflicts [Die 92], i.e structure hazards. Data hazards can be detected by *scoreboard* etc.. The hardware requirements for SOD should be greater than those for SOEC and less than for SOB, because no hardware mechanisms are required to resolve those hazards. Moreover, in many application programs, most hazard information can be statically extracted. If a clever compiler can order the instructions within one thread according to the related pipeline structure, then many hazards can be eliminated statically. Provided that the static information on hazards is integrated in the instruction format, then only simple hardware support is required. The multithreaded processor architecture proposed in chapter 6 employs this strategy.

4.1.2 Multithreaded Multiprocessors

1. Switch on every cycle (MSOEC)[1]

The model MSOEC in multiprocessor environments is mainly used to tolerate the long communication latency which is unavoidable in any von Neumann processor based multiprocessor systems. In comparison to pipeline hazards, such memory latency is usually larger (e.g. more than 100 machine cycles), unpredictable, and grows when the system is enlarged. Therefore, more parallel threads are required to sustain the performance. The best example of this kind of multiprocessor systems is the Denelcor HEP which contains two pipelines (see Figure 3.11). In a configuration with four processing elements 20 - 30 threads (processes in HEP's term) are needed to mask all possible latencies [Jor 85] (both pipeline latency as well as remote memory latency). Other proposed machines with this kind of switching model are Halstead's MASA [Hal 88], Nikhil's P_RISC [Nik 89], Papadopoulos's Monsoon [Pap 91a], TERA [Alv 90] etc.

2. Switch on remote load (SORL)

[1] We use MSOEC to distinguish this model from SOEC in uniprocessor environments.

As mentioned, in a multiprocessor environment multithreading is a technique used to mask the long and unpredictable latency due to remote accesses. It is, however, unneccessary to switch context on every cycle. A context switching is actually neccessary when some operation causing a long latency is encountered. For example, a remote load instruction should cause a context switching, because the successive instructions which use the data from the remote shared memory can only be scheduled for execution when the load operation has completed[2]. The main advantage of this model over MSOEC lies in the better single thread performance, because a thread continues to execute until a remote load instruction is encountered. Thus, fewer threads are needed to cover the long latency, which leads also to fewer hardware contexts required to be integrated in the processor. APRIL employs SORL to improve the single thread performance and to reduce the number of hardware contexts needed to cover the memory latency[Aga 90]. To improve the performance further, SOD can also be introduced for hiding the operation latency at the pipeline level. We call this extended model SORLD (SORL+SOD).

3. Switch on necessity (SON)

To improve the performance of a multithreaded processor, it is important to reduce the frequency of context switching. With SON, a context switching occurs only, when the data neccessary to execute the next instruction is not available. This means that context switches take place on the *use* instructions rather than on the *load* instructions. SON allows a thread to hide some of its memory latency by prefetching data. If a compiler can order a remote load instruction so early that its response is available several cycles before this value is used, no context switching is neccessary for this remote access. Therefore, an obvious benefit of SON is that the frequency of context switching could be reduced, provided that a compiler optimally schedules the execution order of instructions. Another advantage is that several load instructions can be grouped together so that only one *use* instruction needs to wait for the completion of all load instructions of this group together rather than individually [Boo 92]. The cost of this model is that a *use* instruction should be explicitly inserted before instructions which use some data from the remote memory.

4. Switch on explicit instruction (SOEI)

An alternative to SON is to introduce an explicit context switch instruction between the loads and their subsequent uses. The experiments in [Boo 92] have shown that this model can eliminate from 50% - 80% of context switches which are otherwise performed using SORL. Obviously, the performance achieved by SOEI is strongly dependent on the compiler analysis, which reorganizes the remote load instructions and inserts a context switch instruction after

[2]It is worthwhile noting that remote shared stores which do not require acknowledge do not wait for their completion. Therefore, they will not cause context switching.

each group of remote load instructions. The more load instructions are grouped together, the less context switch instructions are required. One of the penalties is that each switch instruction takes a cycle that might have otherwise been used for computation. Further, the code reorganization can lead to some extra overhead (2 - 11 % in [Boo 92]). The work reported in [Boo 92] is, however, encouraging.

4.1.3 Summary

In uniprocessor environments multithreading is mainly used to avoid the problem with pipeline hazards, while in multiprocessor environments this technique is applied especially for masking the long latencies as well as pipeline hazards. For multithreaded machines we prefer to have a small multithreading level, i.e. few threads, because it can reduce hardware cost for integrating hardware contexts (multiple register sets in MASA) in the processor as well as software overhead in managing the activities of threads. Furthermore, a small multithreading level allows to solve problems with little parallelism more efficiently. The classification above also showed the evolution of multithreading models in this direction. The most recent data flow research suggested short threads with explicit switching [Cul 91a]. Short threads execute until their completion at which point some explicit instruction causes a context switch to another active thread. The main advantage of this execution model is that most of the results achieved by experience with dataflow machine can be directly applied to the new model [Cul 91a]. A *clever* compiler is, however, required to extract the sequential instructions from a data flow program and translate it to threads in which the instructions are executed sequentially by using the familiar von Neumann model, while the scheduling of threads for execution is performed with the dataflow model.

4.2 A General Multithreaded Architecture Model

The classification scheme above shows that the differences between the proposed multithreading models lie in their context switching strategies. The following definition is introduced in order to simplify the expression of a program which is executed in multithreading.

> **Definition 4.1:** A **scheduling block (SB)** is a sequence of instructions which are sequentially executed without performing any context switching. The number of instructions executed within a scheduling block is called **run-length** of this scheduling block.
>
> □

In operation, a SB can be initiated when no dependencies to any instruction within the SB exist. This means that once a SB is initiated, it continues execution until its completion without any discontinuity due to any dependency.

A thread may contain one or several scheduling blocks. Scheduling blocks are then the smallest units which the context switching mechanism of a multithreaded processor can manipulate. A SB may be as small as a single instruction (e.g. with the SOEC multithreading model). The run length of a SB depends on the applied context switching strategy.

Another essential parameter for multithreading is latency caused by every instruction which prevents the successive instruction from executing. We extend the definition of latency for multiprocessor environments presented in [Ian 90]:

> **Definition 4.2: Latency** is the time required to complete an operation until its result is available for use as an operand in a subsequent instruction. There are three kinds of latencies:
>
> - **operation latency** O_L which is caused by pipeline hazards in the pipelined execution model,
> - **communication latency** C_L which is caused by accesses to the remote memory or a remote processor, and
> - **synchronisation latency** S_L which is caused by synchronizing the shared variables for parallel threads.
>
> □

It should be noted that the communication latency is not just the communication network delay, it should also contain the time required to proceed the request and return the response just like *rload_handler* and *rstore_handler* in [Nik 92]. So our definition extends the known assumption of the communication latency only as physical network delay. The same applies to the synchronization latency.

The definitions above allow us to define a general multithreaded architecture.

> **Definition 4.3:** A **Multithreaded architecture** is a 5-tuple $< P, L, E, O_s, S >$, where
>
> - P $= < P_s, O_L >$ is an instruction pipeline which is shared by threads running on the architecture, where P_s is the pipeline structure defined in definition 3.3,

- L = $< \{S_L\}, \{C_L, S_L\} >$ consists of synchronization latency, or communication and synchronization latencies,
- E is the environment of multiple thread execution and consists mainly of a set of hardware contexts,
- O_s is the overhead of context switching and typically measured in machine cycles, and
- S: $T_i \rightarrow (T_i(1), ..., T_i(n_i))$, is the scheduling strategy which determines the scheduling blocks $T_i(j)$ of the thread T_i.

□

This definition includes the general properties of different multithreaded architectures. Actually, a multithreaded architecture is an extension of a conventional pipelined processor by introducing the four additional parameters. This is consistent with all multithreading models we mentioned in the last section. The definition is also suitable for multithreaded uniprocessors as well as for multithreaded multiprocessor architectures. For example, in a uniprocessor environment the synchronization latency (but without network delay) should also be considered if multithreading is applied to MIMD computation in which synchronizations are performed through certain shared variables. In a multiprocessor environment the synchronization latency also contains the delay caused by the network, so that some synchronization latencies may be larger than the communication latency. In recent work on the performance analysis of multithreading models [Saa 91], the performance loss due to pipeline hazards is ignored, because the remote latencies are assumed to be much larger than the operation latencies caused by the pipeline hazards. But we insist that operation latencies are an important factor influencing the performance of a multithreaded pipeline, especially if the length of a multithreaded pipeline increases. A multithreaded processor system which only masks the long memory latency can not achieve its optimal performance, because each local processor may suffer from pipeline hazards in the pipelined execution similar to any conventional pipelined processor.

In this architecture model we view a thread as a stream of scheduling blocks. At any time, each thread is in one of four states depicted in Figure 4.2a. A thread continues to execute until some operation causing context switching is encountered. The context switching may occur implicitly (by some hardware mechanism) or explicitly (by some context switching instruction).

Differently, a multithreaded processor is in one of three states (see Figure 4.2b). The context switching may result in extra overhead. In this case, we assume that the execution and the context switching are not overlapped, so that another thread can only begin to execute when the running thread is switched off completely. If there is no active thread, the processor enters the *idle* state until some thread is activated again.

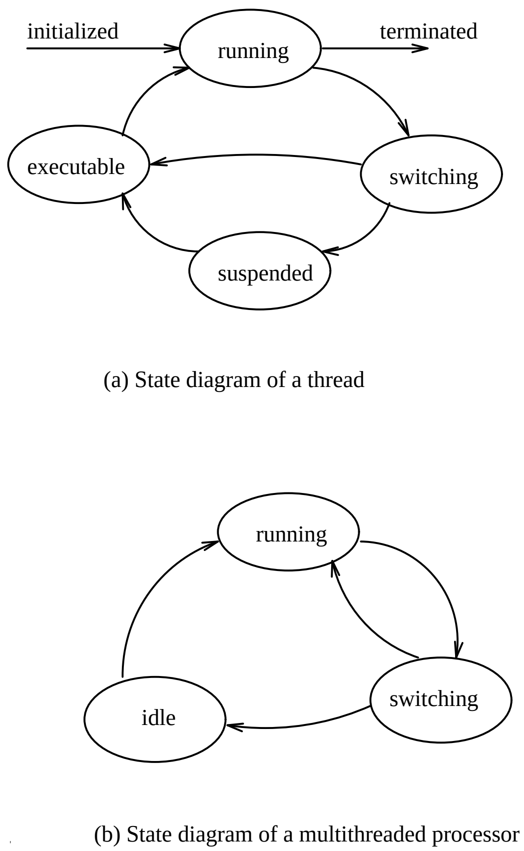


Figure 4.2. States in multithreaded execution

4.3 A Performance Model for Multithreaded Execution

4.3.1 Pipeline Utilization as Performance Metric

As discussed in chapter 3, the objective of pipelining is to improve the execution time of a sequential program, i.e. improve the performance by exploiting the instruction parallelism from single thread (program). The main objective of multithreading is, however, to enhance the machine throughput by exploiting the inter-thread parallelism as well. A good measure of system performance is the socalled *system power*, which is the product of the number of processors and the average processor utilization [Aga 90]. In our architecture model a multithreaded architecture consists mainly of a shared pipeline, together with other mechanisms for supporting multithreaded execution. The processor maintains multiple threads and interleaves them based on some event in order to avoid as many *bubbles* in the pipeline as possible. Therefore, the pipeline utilization can accurately express the processor utilization in a multithreaded environment. Provided that the pipeline utilization takes into account the deleterious effects of factors which influence the efficiency of multithreaded execution, then the pipeline utilization is itself also a good performance measure which reflects the throughput of the machine. In the following discussion we use pipeline utilization $U(n)$ as the measure of performance evaluation, where n is the number of active threads.

Another important measure for performance evalution of a multithreaded architecture is called the *saturation point* defined as follows:

> **Definition 4.4:** The **saturation point** N_{sp} of a multithreaded processor is the minimum number of active threads required to achieve the peak performance of the processor.
>
> □

Obviously, when the number of threads is below the saturation point, there may be no ready thread after a context switch, so the processor will experience idle cycles. A number of threads beyond the saturation point can not lead to any performance improvement, on the contrary, it could degrade the performance due to more possible network conflicts and increased cache interference in a processor configurated with cache memory [Aga 90].

4.3.2 Latencies and Synchronization

The latencies occuring in a general multithreaded architecture can be divided into three categories: O_L, C_L, S_L (see the definition 4.3). In a pipelined processor with static scheduling, the operation latency O_L can be determined statically at compile time, so it is reasonable in our analysis to assume that this latency is constant[3]. The communication latency is mainly dependent on the distance between processors or between processors and memory blocks. Provided that a processor communicates with memory blocks or other processors through some regular interconnection network such as a multistage network and network conflicts are neglected, then such distances are nearly equal and also predictable. The synchronization latency is, however, in general unpredictable, because the synchronization latency is not just an architectural property, it also depends on other factors like scheduling, the computation time of associated threads, program structure, etc.

This study focuses on such application programs in which parallel threads are explicitly initiated and terminated, and programs have a fixed set of threads during the main execution period. This is typical for applications written in extended C or Fortran [Jor 85]. Although synchronization is needed at many points, such synchronization is often explicitly defined with some primitives, and programmers know when and where the synchronization is needed. Thus, we can assume that the synchronization latency has an effect which is only a small fraction of the total computation. We assume further that all programs have a single *fork phase* and *join*

[3]Obviously, the operation latency might be variable in a pipelined processor with dynamic scheduling in which the pipeline will do *out-of-order execution*. In the following analysis, we will not consider this situation.

phase during which all parallel threads are initiated and terminated. We exclude thus any work done outside these phases, because as problem sizes are increased, these overheads become a smaller fraction in the total computation time. These assumptions are consistent with the HEP's program structure with the consideration that *multiple processes should be forked as near to the beginning and joined as near to the end of the program as possible* [Jor 85]. In other multithreading research, similar assumptions were also used [Web 89].

Summarized, two simplifications are introduced in our following discussion:

- If a synchronization occurs in the local processor, no extra synchronization latency will occur, i.e. $S_L = 0$ and
- if a synchronization occurs in a remote memory block or a remote processor, the synchronization latency is equal to the communication latency, i.e. $C_L = S_L$.

4.3.3 Performance of Multithreaded Uniprocessors

Under the assumptions above we can construct a multithreaded uniprocessor model according to definition 4.3 (see Figure 4.3) There is an instruction pipeline consisting of n stages. The scheduling strategy S is defined in the control unit. A thread pool contains the execution environment E where all active threads are resident. The wait buffer is used to store suspended threads. The control unit modifies the states of suspended threads on every cycle and fetches threads from the thread pool for further execution if the related hazards are resolved. Provided that the thread pool is large enough to hold all active threads, i.e., the environment E is large enough, and synchronization effects can be eliminated as mentioned, then the processor model can be described as a triple $< P, O_s, S >$.

The program execution on this model begins with selecting an active thread from the thread pool by the control unit. A thread continues execution until some event is detected by the control unit as defined by the scheduling strategy. The running thread is then switched off and transfered to the wait buffer with an overhead of O_s machine cycles. Subsequently, another thread will be loaded to the instruction pipeline. The control unit monitors the execution status in the pipeline and registers the status of all suspended threads in the wait buffer. If any thread in the wait buffer is ready for execution, this thread will be fetched from the wait buffer, routed to the thread pool, and wait for its next turn to execute.

A *critical* operation in this model is to determine whether an assumed event has occured. Such an event can often be detected after the fetched instruction has been decoded or its operation has been performed. If such an event can only be detected at the pipeline stage i, then the

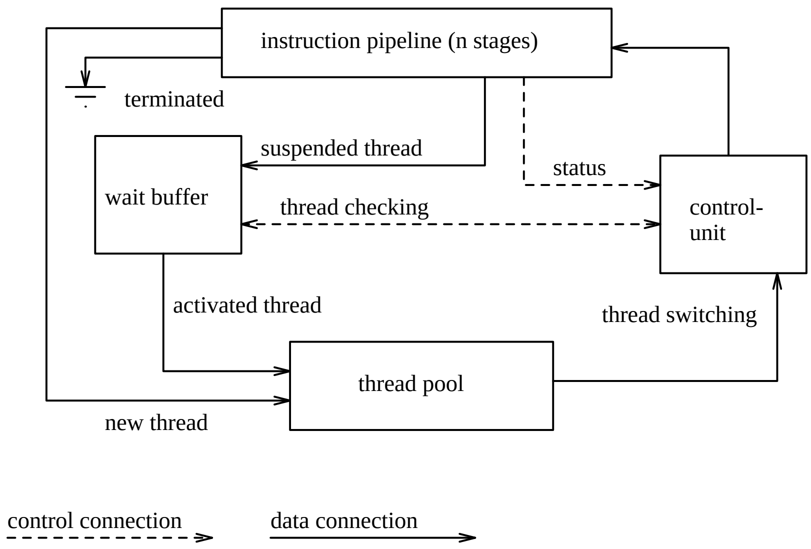


Figure 4.3. Architecture model of a multithreaded uniprocessor

other $i-1$ subsequent instructions from the same thread must be ignored. This delay is called **decode latency**, which can be viewed as a fraction of the context switch overhead. But we must notice the essential difference in scheduling strategies between a conventional pipelined processor and a multithreaded processor. In a multithreaded processor, a new thread will be scheduled for execution immediately after an event (for example, a pipeline hazard) is detected, whereas in a pipelined processor a next instruction can be issued for execution only when the previous hazard is resolved. We shall present a possible solution to avoid the decode latency in our proposed architecture in chapter 6.

Performance of SOEC

With SOEC the processor fetches only one instruction from a thread (say thread i) and switches to another thread j with the overhead of O_s cycles[4]. The switched thread i is enabled again after $P_l - 1 - O_s$ cycles, where P_l is the length of the pipeline, namely, once the running instruction leaves the last stage of the instruction pipeline. Assuming that each thread has N instructions, thus for n threads being executed on the processor in parallel, the utilization of the pipeline is as follows:

$$U(n) = \frac{Nn}{P_l - 1 + N(n(1+O_s)+I)} \tag{4.1}$$

where $P_l - 1$ is the setup time for the first instruction of the first thread entering the pipeline, and I the idle time occuring if not enough parallel threads are available to hide the operation

[4] With SOEC the overhead O_s is the distance between two consecutive threads entering the pipeline, therefore, $O_s \leq P_l - 1$. O_s may be zero, i.e. there is no overhead for context switching.

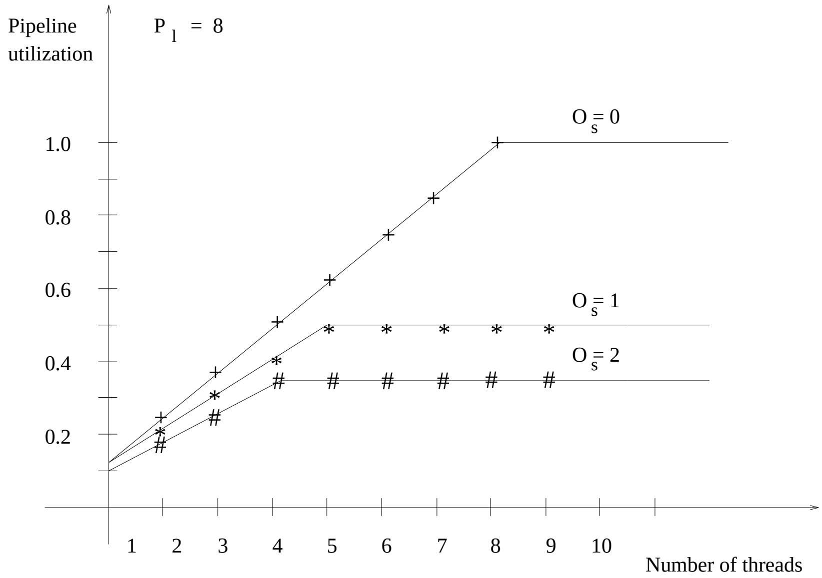


Figure 4.4. Performance of SOEC

latency[5]. I is determined by

$$I = max\{0, \quad P_l - n(O_s + 1)\} \tag{4.2}$$

Hence, $I > 0$ means that some idle time occurs during the thread execution. The operation latency will be hidden if sufficient threads are available. Then, the pipeline utilization combining these two different cases is ($N \gg P_l, n$):

$$U(n) = \begin{cases} \frac{1}{1+O_s} & \text{if} \quad n(O_s + 1) \geq P_l \\ \frac{n}{P_l} & \text{otherwise} \end{cases} \tag{4.3}$$

The first term is the utilization at the saturation point. It is obvious to get the saturation point from the expression above

$$N_{sp} = \frac{P_l}{1 + O_s} \tag{4.4}$$

Figure 4.4 shows the performance with some different context switch overheads. Note that it is common that no extra overhead is required to switch context with SOEC, because the processor switches to another thread regardless of any hazard detection. This is true in the case of HEP and other machines such as MASA, P-RISC, etc.. Provided $O_s = 0$, the utilization form can be simplified to

[5]With SOEC each instruction has an operation latency of $P_l - 1 + O_s$

$$U(n) = \begin{cases} 1 & \text{if } \; n \geq P_l \\ \frac{n}{P_l} & \text{otherwise} \end{cases} \tag{4.5}$$

with the saturation point $N_{sp} = P_l$.

Performance of SOD

As mentioned, with SOD a thread continues execution until a certain hazard is detected. In this case the successively fetched instructions must be eliminated, which leads to so-called *decode latency.* To simplify the discussion, we assume that all threads have the same instruction distribution.[6] Further, all hazards lead to equal operation latency. Without considering the structure hazards, the average run length of a thread is

$$R = \frac{1}{p_c + p_d} \tag{4.6}$$

where p_d und p_c have the same meaning as in section 3.2. This means that the processor switches to another thread after the execution of R instructions. Therefore, it is easy to get the utilization for n active threads

$$\begin{aligned} U(n) &= \frac{Nn}{P_l - 1 + N/R(n(R + O_s) + I)} \\ &= \frac{Nn}{P_l - 1 + N(n(1 + O_s/R) + I/R)} \end{aligned} \tag{4.7}$$

where the idle time I is determined as follows

$$I = max\{0, \quad O_L - n(R + O_s) + 1\} \tag{4.8}$$

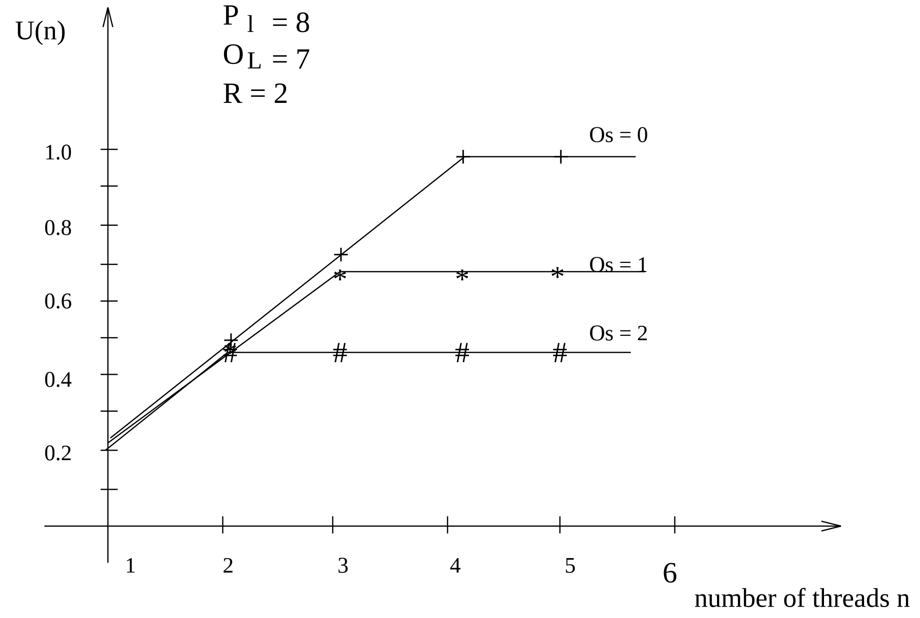
where O_L is the length of the operation latency. Provided that the total length of threads N is much larger than the other parameters, the above expression can be further simplified to

$$U(n) = \begin{cases} \frac{1}{1+O_s/R} & \text{if } \; n(R + O_s) - 1 \geq O_L \\ \frac{Rn}{1+O_L} & \text{otherwise} \end{cases} \tag{4.9}$$

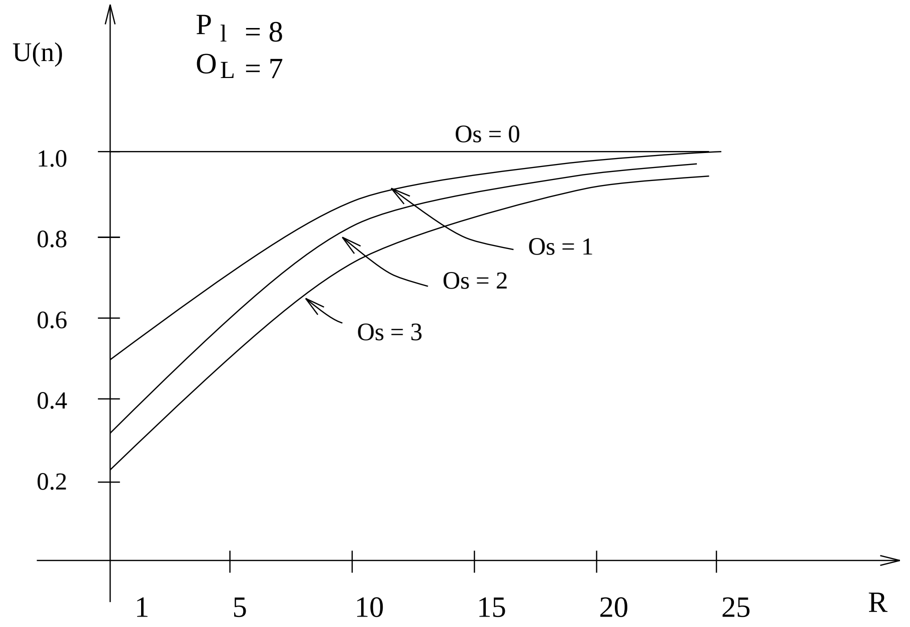
with the saturation point

$$N_{sp} = \frac{O_L + 1}{R + O_s} \tag{4.10}$$

[6] Actually, this assumption can be improved with the more realistic assumption of the run length behaving as some stochastic process which has a certain distribution as in [Saa 91].



(a) U(n) as the function of n and Os



(b) U(n) as function of R and Os at saturated point

Figure 4.5. Performance with SOD

As expected, the *saturated* performance depends only on the proportion of O_s and R. Figure 4.5. shows the relationship between these parameters. The larger the overhead of context switching is, the less threads are required to hide operation latency, but also the less peak performance. To improve the throughput of the machine, it is desirable to reduce or completely eliminate the context switch overhead. Another influence on the performance is the length of operation latency which can be improved with a small hardware cost, for example, by integrating bypass ways between functional units.

The decode latency may be the main source of context switch overhead, because most of the discontinuities in pipeline operations can only be determined after the fetched instruction has been decoded, and successive instructions will be executed only after the corresponding hazard has been resolved. A direct solution to the decode latency is to save all fetched instructions from the same thread in the stages where they are and make them executable after the previous hazard is resolved. For the discontinuities caused by branch instructions, the fetched instructions must often be thrown away if the actual branch direction is outside these instructions. We will present a simple solution to this problem in our proposed multithreaded architecture in chapter 6.

Further, the average run length of a thread R influences the performance greatly. R depends on the application as well as the hardware structure. In a deeply pipelined processor R could become small (e.g. less than two) if the application does not contain much regular data structure and parallelism.

4.3.4 Performance of Multithreaded Multiprocessors

Figure 4.6 shows the execution model of a multithreaded multiprocessor configuration. Differently from the model for a multithreaded uniprocessor (see Figure 4.3), the second latency, i.e. communication latency due to accesses to the remote memory blocks or other processors, must be considered. To handle such remote accesses, the following will be performed: when a remote access is encountered, the multithreaded pipeline builds a message (request) which includes, for example, a function field (read/write), a memory address, an identifier of the issuing thread etc., sends it to the network, and then suspends the running thread into the wait buffer. After receiving the response, the synchronization unit activates the corresponding thread and routes it to the thread pool again.

Performance of MSOEC

A remote access results in a longer latency compared to normal pipeline hazards. The more remote accesses there are in a program, the more parallel threads are required to fill the instruction pipeline fully. In the following discussion, let R_r be the average distance (i.e. the

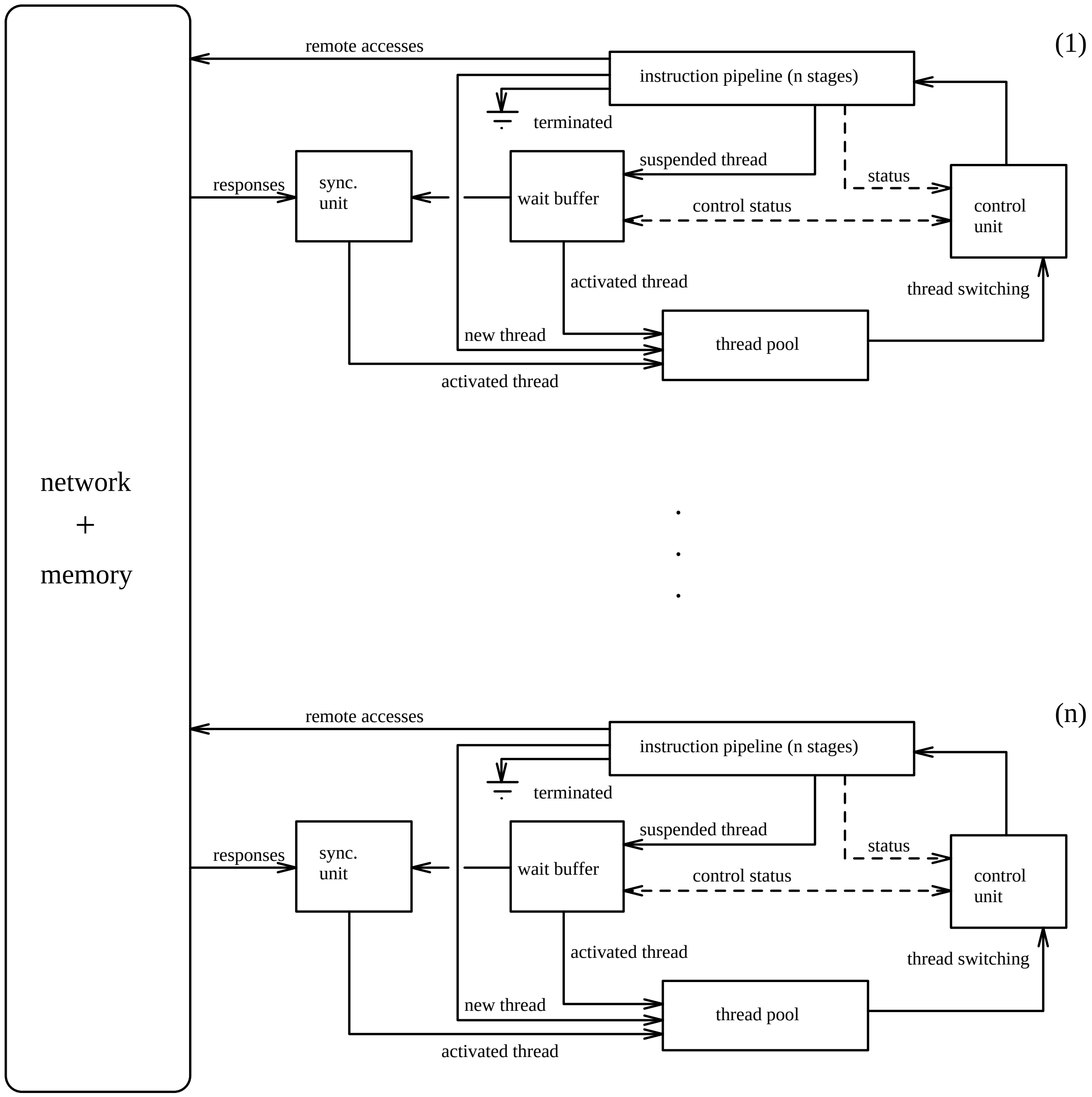


Figure 4.6. Architecture model of a multithreaded multiprocessor

number of successive instructions) between two remote accesses from a single thread. Assume that p_r is the probability of executing one instruction that will trigger a remote access, then $R_r = 1/p_r$.

If a multithreaded processor is applied as the processing element in a multiprocessor environment, first of all it is necessary to mask the long communication latency by exploiting program parallelism in order to reduce or completely avoid the associated loss in performance. MSOEC masks both communication latency and pipeline hazards, provided that enough parallelism exists in the program.

In the case of a single thread, the performance loss with MSOEC is obvious. $R_r - 1$ out of R_r instructions are recirculated in the instruction pipeline directly after execution, and the last one will trigger a remote access which is routed to the network. This results in performance:

$$U(1) = \frac{R_r}{(R_r - 1)P_l + P_l + C_L} = \frac{1}{P_l + C_L/R_r} \tag{4.11}$$

In the case of multiple threads, however, an exact computing of U(n) becomes more complicated, because at any time some threads are suspended while their requests of remote accesses are flowing in the network, and others are executed in the pipeline. Further, the number of threads (both active and suspended) are dynamically changed in the whole execution period. The processor reaches its saturation point only when both communication and operation latencies are hidden.

A simplification of the behaviour of thread execution is as follows: at the saturation point the processor reaches a stable state in which the number of active threads n_a and suspended threads n_s are constant. For each active thread the duration until the next remote access is thus $n_a(1+O_s)R_r$, where $n_a(1+O_s)$ is the duration until the execution of the next instruction (not remote instruction) from the same thread. Therefore, we can assume that the average distance between two remote accesses to the network is:

$$D_r = (1 + O_s)R_r \tag{4.12}$$

The saturation point with the associated U(n) then is:

$$N_{sp} = N'_{sp} + \frac{C_L}{D_r} = \frac{R_r P_l + C_L}{R_r(1 + O_s)} \tag{4.13}$$

where N'_{sp} is the same as the expression 4.4 for SOEC, which is required to hide local latency, and $\frac{C_L}{D_r}$ the number of issued requests to the network, which are required to mask the long communication latency.

For U(n) under the saturation point the behaviour of thread execution is more complicated, because the duration between two successively executed instructions from the same thread depends on the number of active threads, which relies further on other factors. For example, the history of each active thread, the rate of triggering remote accesses to the network, and the rate of the responses returning from the network, etc. Therefore, the number of active threads might vary at each execution cycle. We will present some simulation results in the next section.

Performance of SORL (Switch on Remote Load)

With SORL the processor continues executing a single thread until a remote load instruction is encountered. The controller of the processor switches to another active thread, while the pipeline serves the remote request. In most of those processors the instruction pipeline is built on basis of conventional pipelined processors like in [Aga 90], hence the pipeline hazards must be handled in such processors. Provided that all pipeline hazards are covered by the same number of **bubbles** b_p, the distance between two remote loads to the network is then

$$D_r = R_r + \frac{R_r}{R} b_p + O_s, \tag{4.14}$$

where R is the distance (i.e. the number of instructions) between two successive pipeline hazards. Therefore, at its saturation point the pipeline utilization U(n) is

$$U(N_{sp}) = \frac{1}{1 + b_p/R + O_s/R_r} \tag{4.15}$$

at

$$N_{sp} = \frac{C_L + P_l}{D_r} = \frac{(C_L + P_l)R}{RR_r + R_r b_p + RO_s}. \tag{4.16}$$

where $C_L + P_l$ is the total latency for a remote load.

SORLD: an Extension of SORL through SOD

If the SOD is introduced into the SORL execution model, all pipeline hazards can be completely masked, i.e. multithreading is applied at two levels: intra-thread and inter-thread, to mask pipeline hazards as well as communication latencies, respectively. We distinguish here between the overheads for the thread switching O_p due to pipeline hazards and the thread switching O_r due to remote accesses. The reason is that different operations are performed for these two context switches. Thus, actual implementations may result in different overheads.

The distance between two remote loads becomes

$$D_r = R_r(R + O_p)/R + O_r \tag{4.17}$$

and

$$U(N_{sp}) = \frac{1}{1 + O_p/R + O_r/R_r} \tag{4.18}$$

at the saturation point

$$N_{sp} = N'_{sp} + \frac{C_L}{D_r} = \frac{O_L + 1}{R + O_p} + \frac{C_L R}{R_r(R + O_p) + O_r R}. \tag{4.19}$$

The improved execution model SORLD has an obvious advantage over SORL, especially for a deep pipeline where the operation latencies are also correspondingly larger. The meaning of the SORLD model is that it is easier to reduce the overhead for context switching or overcome it completely than to reduce or overcome the pipeline hazards in a deep pipeline. One penalty of the SORLD model is that it requires more active threads than SORL to fill the pipeline fully, but achieves a better peak performance.

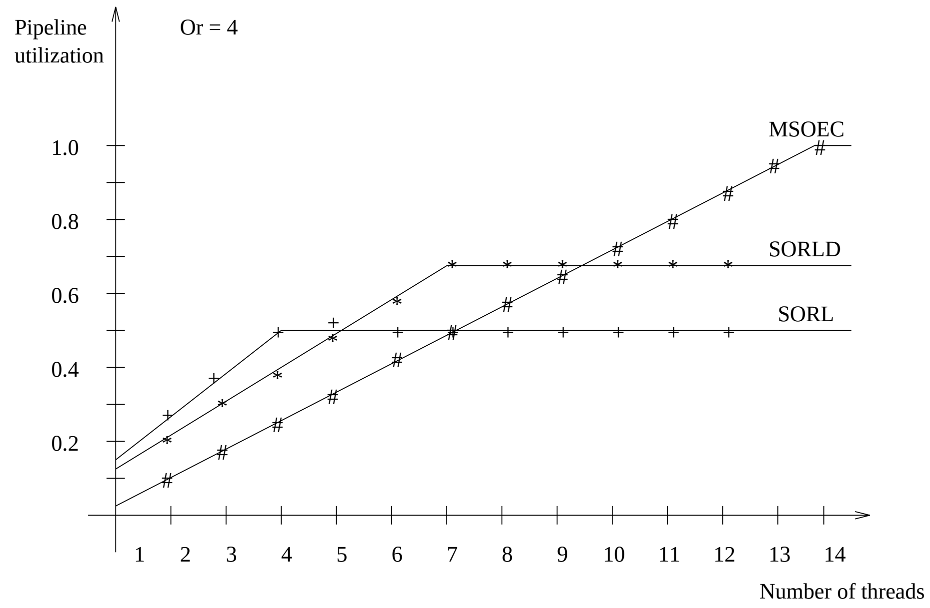
Analysis

The performance expressions above contain several parameters which can be used to define different architecture configurations. One realistic configuration extracted from several publications is as follows:

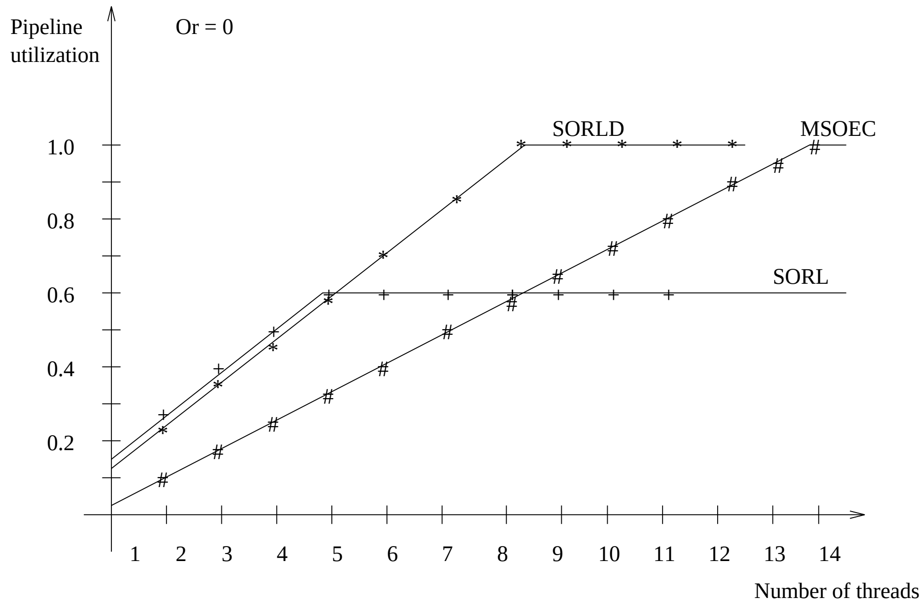
$< P_l = 8, O_L = 7, b_p = 2, R = 3, O_p = 0, R_r = 12, O_r = \{0 - 4\}, C_L = 64 >$

For example, in [Cul 91a] it was reported that the average distance between two *split-phase* operations lies between 10 and 15 instructions, this corresponds here to $R_r = 12$. The communication latency $C_L = 64$ cycles is suitable for a multithreaded multiprocessor with a medium number of processing elements. In a larger configuration the communication latency will be enlarged correspondingly (e.g. several hundred cycles as assumed in [Boo 92]). O_p might be equal to 0 in a processor with several duplicated hardware contexts, while O_r is the overhead for constructing a remote request, switching out a suspended thread and switching another active thread in the pipeline. As in [Aga 90] we assume O_r lies between 0 and 4. O_r becomes zero only when the above task is performed by some special hardware mechanism or a special processor [Nik 92].

Moreover, it should be noted that both O_L and b_p belong to the operation latency due to pipeline hazards. However, there is an obvious difference between O_L and b_p in our analysis. O_L is the time required between switching the running thread and making this thread executable



(a) Analytical performance of mulithreaded multiprocessors (1)



(b) Analytical performance of mulithreaded multiprocessors (2)

Figure 4.7. Analytical performance of multithreaded multiprocessors

again. O_L is 7 machine cycles, because we assume that a suspended thread will become executable after the instruction causing context switching leaves the last stage of the instruction pipeline. This means that no hardware mechanism is assumed to reduce the operation latency for a *pure* multithreaded processor. On the other hand, we treat b_p as the average number of *bubbles* required to be inserted to resolve pipeline hazards. Because most multithreaded processors with SORL employ both software and hardware mechanisms to reduce the effects of pipeline hazards, the actual value of b_p is often small, for example, the improved DLX integer pipeline has a value b_p of one machine cycle [Hen 90].

Figure 4.7 shows the analytical performance of different multithreaded execution models on this configuration. As seen, MSOEC has a poor performance if not enough parallelism exists to hide pipeline hazards as well as the long communication latency. SORL requires a small number of threads to achieve its peak performance, which is equal to 0.6 even if no overhead for context switching is encountered (Figure 4.7b). A better tradeoff can be achieved if the pipeline hazards are also hidden using multithreading, i.e. SORLD. We see that SORLD reaches the same peak performance as MSOEC, but with fewer threads (provided O_r is also equal to 0 as in MSOEC). The performance of SORLD could be further improved if the *run length* of threads could be enlarged. This is actually the goal of proposing other multithreading models such as SON and SOEI. We can get the associated performance for SON and SOEI by enlarging the distance of successive remote accesses, because the main effects of these models are the reduction the frequency of context switching by prefetching data and eliminating some unnecessary remote accesses. An exact estimation of the performance for these models is only possible if some optimal scheduling strategy for corresponding compilers is available. How to construct such a compiler and to evaluate its effects are, however, beyond the scope of this work.

A simulation model

The analytical expressions for the performance of different multithreaded execution models have been achieved by simplifying the thread execution behaviour and by introducing some static parameters. In practice, the performance behaviour would be more complicated in a multithreaded multiprocessor environment. As mentioned, the exact behaviour of thread execution under the saturation point is difficult or impossible to describe by some simple mathematical equation.

In order to observe the actual behaviour of thread execution, a simulation program was written in C on a Sun-SPARC workstation according to the execution model shown in Figure 4.6. The simulated processor may have any architecture configuration defined with the parameters above. The simulator allows to vary all parameters and simulates the process of thread execution in machine cycles. Some important statistics are collected: U(n), the dynamic number of active threads, and the distribution of remote requests.

For comparison with the analytical results above, the simulation was performed with the following constant parameters:

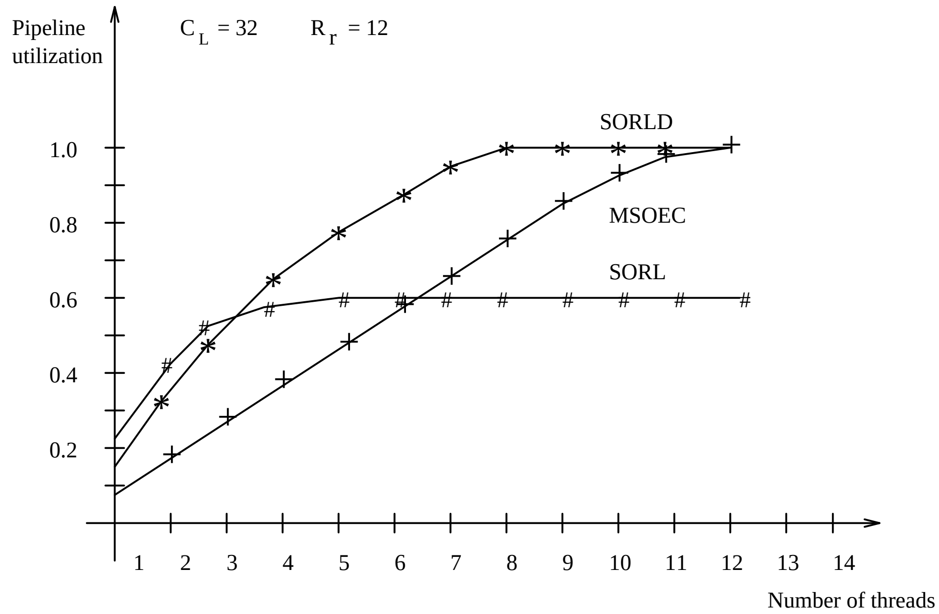
$$< P_s = 8, O_L = 7, b_p = 2, O_p = 0, O_r = 0 >.$$

To obtain more accurate execution statistics, we assume that the parameter R is uniformly distributed between 2 and 4 (with an average value of 3) during the whole execution time. For the parameter R_r we chose two average values 12 and 24 in order to observe the associated effects, because this can reflect the influence generated by other improved execution models such as SOD and SOEI. Similar to R, we assume that R_r is uniformly distributed between 8 and 16 or between 20 and 28 (with the average values of 12 and 24, respectively). No dynamical effects such as synchronization between parallel threads are considered.

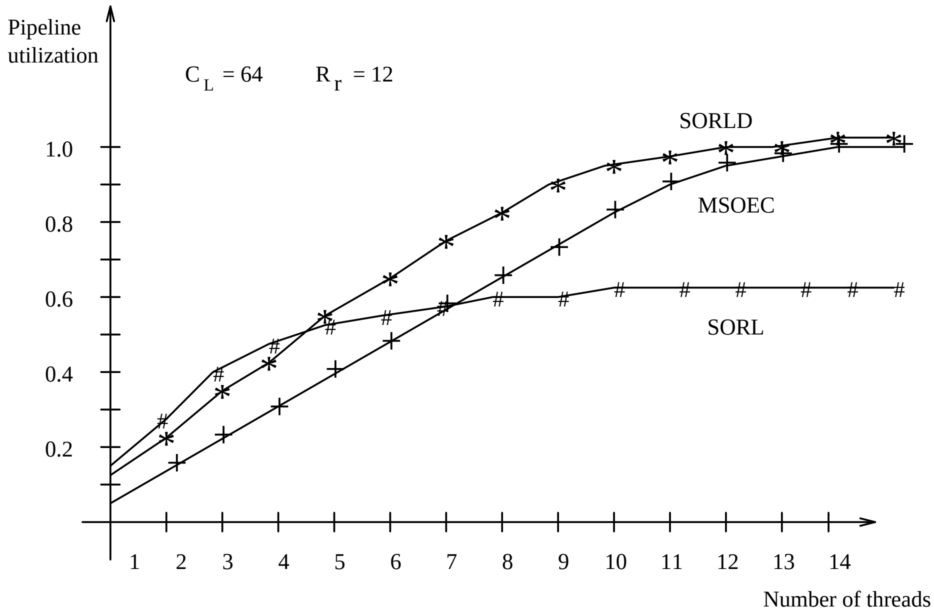
Each configuration was simulated for 10,000 machine cycles, after a startup period of 200 cycles. The skip of this startup period is necessary due to the long memory latency. Further, the execution data of each thread was independently generated according to the assumptions above (i.e. R, R_r, and C_L), and no synchronization between different threads was assumed. Note that the pipeline utilization reflects the efficiency of the simulated processor configuration for the whole execution phase and therefore is computed through the useful cycles in the pipeline divided by the whole required machine cycles. The main results are shown in the Figure 4.8 and 4.9.

The performance in Figure 4.8 shows the variation due to enlarging the communication latency, while Figure 4.9 shows the performance variation due to enlarging the remote access distance. A longer run length results in fewer threads required to fill the instruction pipeline.

The experiments above show that a well designed multithreaded processor should exploit both intra-thread and inter-thread parallelism to tolerate latencies at two different levels, respectively. Furthermore, to reduce the number of parallel threads, the inter-thread parallelism should only be extracted when no intra-thread parallelism exists. However, a multithreaded processor based on the conventional pipelined architecture could reintroduce the complexity of conventional pipelined processors and require high hardware cost. As proposed in the last section, the SORLD model is an effective trade-off between MSOEC and SORL models. One of the requirements for processors based on SORLD is that some hardware mechanisms must be provided to switch the running threads out at certain pipeline stages whenever a certain discontinuity at those stages is detected. A simple but effective mechanism is employed in the processor presented in chapter 6.

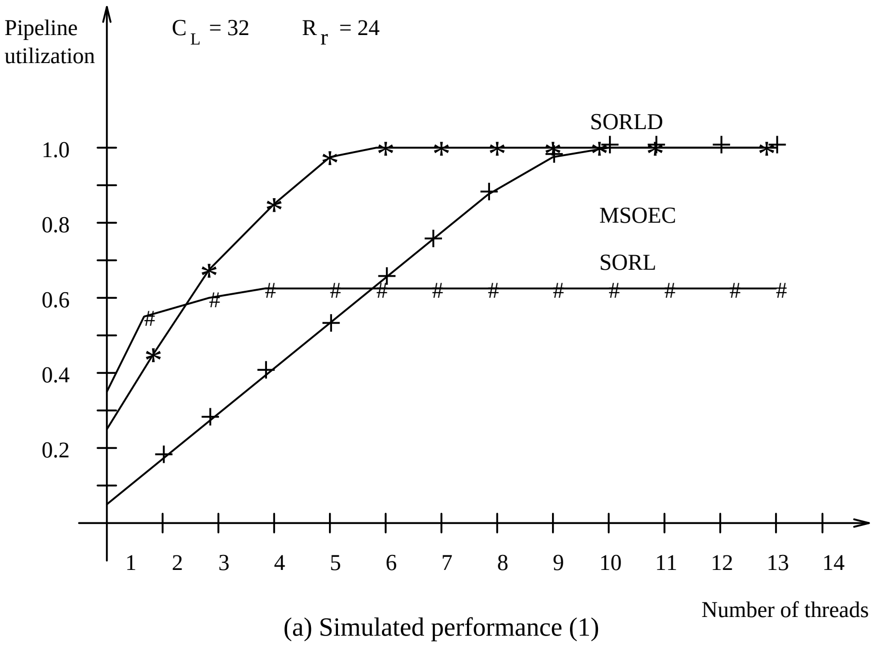


(a) Simulated performance (1)

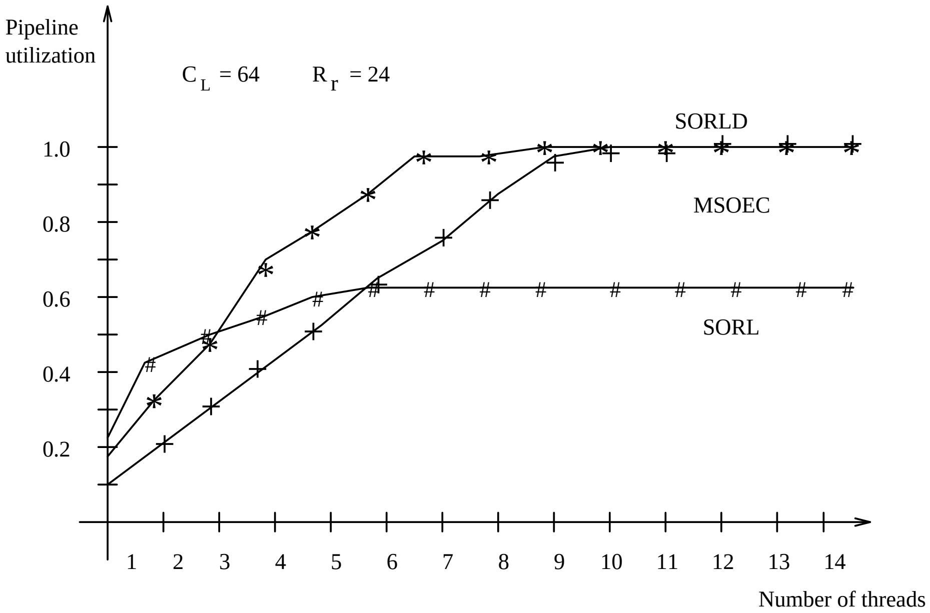


(b) Simulated performance (2)

Figure 4.8. Simulated performance with varying the communication latency



(a) Simulated performance (1)



(b) Simulated performance (2)

Figure 4.9. Simulated performance with varying the run length

4.4 A Realistic Multithreaded Processor

As the next experiment to evaluate performance of multithreaded execution and different scheduling strategies, we construct a multithreaded architecture model based on a well known RISC processor – DLX [Hen 90]. It is often cost effective to construct a multithreaded processor based on a well known processor like DLX. Such multithreaded processors cover most of the proposed architectures like APRIL, P_RISC, MASA etc. Moreover, this experiment provides the performance estimation of possible multithreading strategies. Such estimation is important for designing our multithreaded processor presented in chapter 6.

4.4.1 Architecture Model

As with any multithreaded processor architectures, there must be certain mechanisms to maintain several threads in the processors and schedule executable threads according to scheduling strategies. According to our multithreaded architecture model defined in section 4.2, we extended the basic DLX pipeline to a multithreaded processor by introducing the following mechanisms: an extra stage *thread select* and a *thread pool* (see Figure 4.10). The extended multithreaded pipeline then consists of at least six stages. The main extensions of the DLX-pipeline are included: (1) the stage IF does also partial decode of the just fetched instruction and sends related control signals to the stage TS for the purpose of context switching; (2) in multiprocessor environments the stage WE forms the communication message for remote accesses. The purpose of introducing the partial decode for the fetched instruction is to eliminate the *decode latency* mentioned in the last section, i.e. to detect the event causing context switching through such simple partial decode. The multithreaded pipeline would be lengthened if a complex ALU were included. The conceptual pipeline structure is shown in Figure 4.10. The arcs from the pipeline stages to the thread pool mean that the TD (thread descriptor) of some suspended thread is inserted into the thread pool again after the execution of this stage. Here we only are interested in the logic behaviour of the thread pool. In practice, such a thread pool may be a memory unit with maximum four write ports and one read port.

As in the last section, we define a *thread* to be an instruction stream consisting of a collection of *scheduling blocks* (SBs). In our architecture model, scheduling blocks are dynamically determined by the stage *thread select* TS according to certain scheduling strategies. The instructions within one scheduling block are executed sequentially in the pipeline. The stage *thread select* schedules an enabled thread for execution when a context switch occurs. Each thread is characterized by a *thread descriptor* (TD) which contains the necessary information for thread execution, e.g. instruction pointer, frame base address etc.. The *thread pool* holds all thread descriptors of executable threads.

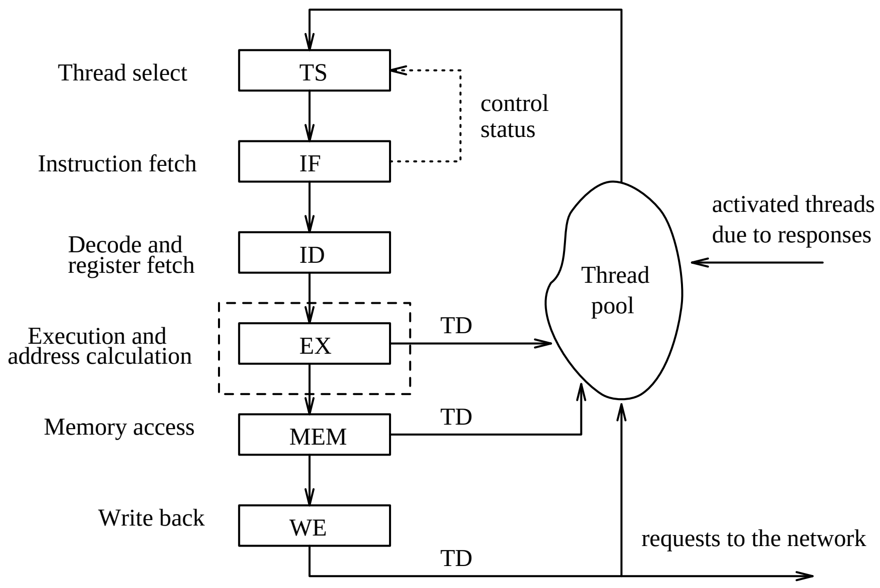


Figure 4.10. Architecture model of a realistic multithreaded processor

Since the multithreading technique interleaves the execution of different threads to avoid pipeline hazards and other latencies instead of using some complex hardware mechanism, no special hardware supports for resolving pipeline hazards are assumed for our architecture model. We assume also that there is no overhead for context switching. This is true for many multithreaded processors such as HEP, MASA, etc., but in conflict with other machines such as APRIL, which assume a cost of several cycles for cleaning the processor pipeline[7].

Multithreaded architectures maintain the state of several threads and switch among them based on some event. If we do not consider the effects of cache misses and exceptions, we can simplify the thread scheduling in our architecture model: the processor switches among several executable threads based on the execution of certain instruction, e.g. the context switching occurs when an instruction causing some hazard is detected. Therefore, here we define several different scheduling strategies according to the execution of different instructions. This is also consistent with most of the proposed multithreaded processors.

For our experiment, we used the same instruction distribution as in table 3.1, but we do not account for some special instructions for thread initiation and synchronization whose effects are a small fraction of the total computation and therefore can be neglected. Four possible scheduling mechanisms (also called dispatching mechanisms) for the multithreaded uniprocessor are examined:

- *Switch on every cycle (SOEC):* with this mechanism only one instruction of each enabled thread is allowed to be executed at any time; each stage of the pipeline contains an

[7]Each APRIL processor contains a cache to reduce remote memory accesses. Context switching occurs when a cache miss is detected. This occurs at the late stage of the instruction pipeline, therefore, some cycles are needed to eliminate the fetched instructions in the pipeline.

instruction from a different thread.

- *Maximum pipeline delay (MPD):* in this case, we consider the worst inter-instruction delay required for correct execution; maximum delay cycles are inserted after each control and data-manipulation instruction.

- *Static pipeline delay (SPD):* here the compiler generates the static information of data dependencies for ALU and Store instructions. This delay information is coded in each ALU and Store instruction.

- *Control and load delay (CLD):* here we assume that all data dependencies of ALU and Store instructions are avoided in hardware by fully internal forwarding, hence, only control and load dependencies are considered.

It should be noted that only CLD requires extra hardware support for detecting hazards and forwarding results directly. Further, load dependencies are not included in SPD, since they are similar to control dependencies which can not completely be eliminated with some simple hardware mechanism. In this analysis, we focus on the different scheduling mechanisms for our extented multithreaded architecture and ignore other issues such as cache misses, variable operation latencies etc.

4.4.2 Simulation Results

A simulation program at the instruction level was written to evaluate its performance issues. Two important time intervals are used in simulating the different scheduling mechanisms: *control latency* and *data latency* (see definition 3.2). Without branch prediction the control latency is six cycles in the extended multithreaded pipeline with eight stages (three execution stages), because each jump destination is computed at the stage MEM. Without internal forwarding the maximum data latency is equal to five cycles in the same pipeline.

The simulation has been performed to illustrate the different performance of four possible mechanisms under the following assumptions:

1. The execution stage EX consists of three substages EX1, EX2, EX3, so the instruction pipeline has 8 stages all together. Results for a multithreaded processor based on the basic DLX pipeline (6 pipeline stages) were reported in [Fan 92b].

2. Enabled threads are chosen at random for scheduling.

3. The distribution of executed instructions is derived from several general-purpose codes as shown in table 3.1.

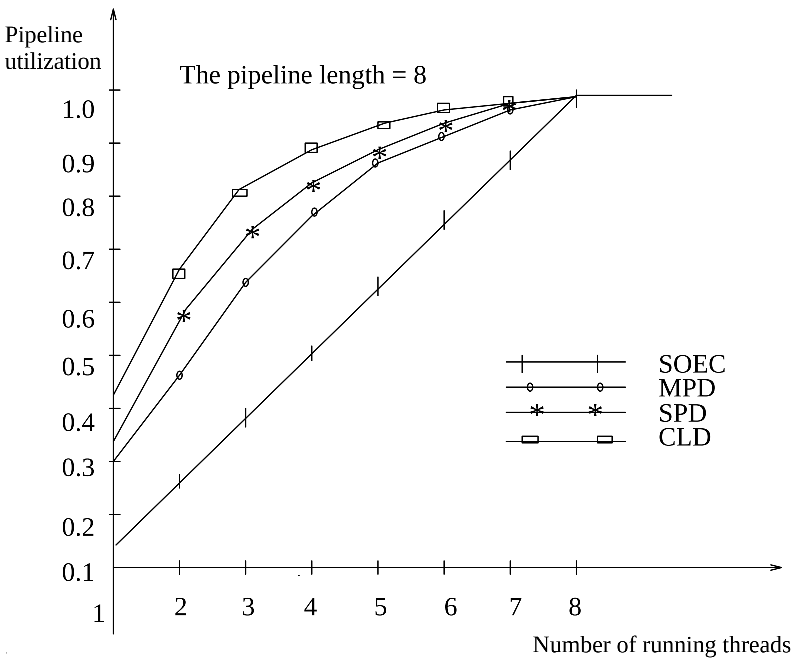


Figure 4.11. Simulated performance of a realistic multithreaded processor(1)

4. The data latency in SPD is uniformly distributed over the range from 0 to 5.

5. No structure hazards, variable operation latencies and cache misses are considered.

We lengthen the EX stage to achieve the same number of pipeline stages assumed in the last section. Further, it is also consistent with many proposed multithreaded processors (HEP, MONSOON etc.) and advanced RISC processors (Alpha, MIPS4000 etc.). For each configuration 10,000 machine cycles were simulated under the assumpltions above. The simulated performance for the uniprocessor is shown in Figure 4.11.

As expected, CLD achieves the highest performance of the four different models. It is interesting to see the performance differences between MPD, SPD and CLD, which represent three different implementation methods. MDP is easy to realize, because no hardware support is required. The compiler checks all possible data dependencies and inserts the maximum number of *no-ops* after each instruction causing data dependencies to successive instructions. With SPD, the compiler must determine the accurate operation latency due to data dependencies and insert necessary no-ops as needed, whereas CLD needs more hardware costs to integrate some hardware mechanisms for detecting pipeline hazards at run time and forwarding results directly. A trade-off between these models depends mainly on the applications. If the processor is mostly used for applications in which much regular parallelism can be extracted, SPD is more attractive. For applications with little parallelism, CLD delivers a better performance at the cost of more hardware.

In further simulations we included the long latency due to remote accesses in order to observe the performance issues in multiprocessor environments. The further assumptions are:

- Context switching occurs also when a remote load instruction is decoded.
- The frequency of executed remote load instructions is 10%, which is derived from the experiments in [Cul 91a].
- The remote latency is predictable. Therefore, no synchronizing loads are considered, which lead to unpredictable latencies.

Again, for each configuration 10,000 machine cycles were simulated in multiprocessor environments. Figure 4.12 shows the performance of four different scheduling strategies in a multithreaded environment with a remote latency of 64 cycles. The careful reader will notice that the performance curves depicted in Figure 4.12 are more pessimistic than those shown in Figure 4.7. The main reason for degraded performance compared to the analytical result lies in choice of simulation parameters which are extracted mainly from several complex multithreaded processors, in which some hardware as well as some optimal scheduling policies are exploited. For example, we chose the distance of two pipeline hazards $R = 3$. It is realistic for an application provided the compiler can make some data dependency analysis and reorder the sequence of instruction so that some hazards can be eliminated by, for example, prefetching related operands. If we examine the four possible scheduling strategies on the DLX based multithreaded pipeline again, we can see that this architecture model represents more realistic features of a simple RISC based multithreaded processor. A RISC processor can typically not completely eliminate the pipeline hazards due to control and load dependencies. Instead of assuming some hardware or software mechanisms to reduce the effects of pipeline hazards, we assume only that all of those hazards are hidden by multithreading in order to observe the gain offered by the multithreaded execution. Therefore, the average distance of two pipeline hazards is typically less than three. With the most efficient scheduling strategy CLD, we have an average distance $R = 1/(0.2 + 0.15 + 0.08) \approx 2.3$, which is directly concluded from table 3.1.

4.5 Summary and Remark

This chapter has described the main issues concerning the performance of typical multithreaded processors. Through a classification of different multithreading models, some common properties of those models have been extracted and then a general multithreaded architectural model has been defined, which was used as basis for the further analysis. The analytical and simulation results presented the essential performance characteristics of multithreaded execution. The requirements for a fast uniprocessor and for an efficient multiprocessor led to some

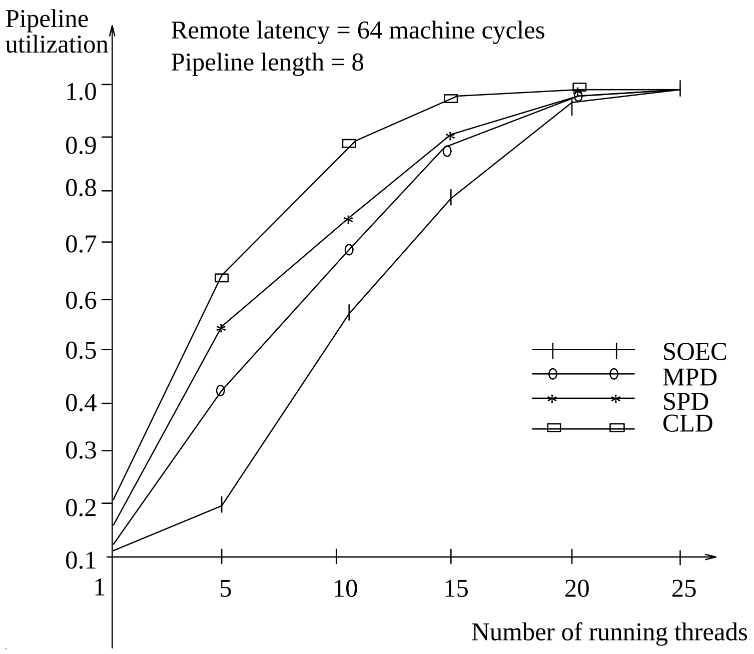


Figure 4.12. Simulated performance of a realistic multithreaded processor(2)

new problems which can not be efficiently solved with conventional methods. Our analysis in this chapter has shown that multithreading as a technique to handle such problems both for uniprocessors and for multiprocessors is attractive and cost effective.

The limitation of our performance analysis for multithreaded processors may be the neglect of performance effects due to synchronization latency between threads. This is a complicated issue, because synchronization latency is a dynamic factor which depends on many other factors as mentioned in section 4.3.2. An exact estimation of synchronization effects on performance can be achieved only by simulating program execution on a concrete multithreaded architecture. Further, such results may be strongly dependent on the scheduling strategy, compiler's ability to eliminate unnecessary synchronization points, and so on. The effects of synchronization latency on processor performance is also an issue which will be pursued in the future.

Chapter 5

Fundamentals of the Latency-directed Multithreaded Execution (LDME)

In the last chapter, the features and performance advantages of multithreading have been discussed. To exploit such advantages efficiently, a proper execution model is necessary. In this chapter, some fundamental requirements on such a model will be outlined. After this, a multithreaded execution model will be presented, which contains these fundamental features. Its architectural support and associated hardware implementation will be described in the next chapter. As this work focuses on the performance and architecture issues of multithreading, no software issues such as language semantics or compiler support will be dealed with in detail. But the proposed model contains the primitives which are fundamental for the implementation of a concrete language.

5.1 Requirements on Multithreaded Execution Models

Based on observations on performance advantages of multithreading presented so far and on various studies of multithreaded architectures, the following conclusions on requirements of an efficient multithreaded execution model are drawn:

- A multithreaded execution model should be a MIMD model with a dynamic model of storage. With the MIMD model a program is expressed as a *partial ordering* of instructions, which allows to exploit parallelism both within an instruction thread and between instruction threads. A dynamic model of storage is necessary for a general MIMD model, because the execution of tasks (code blocks) form a tree, and the form of this tree cannot be determined statically [Cul 91a] [Pap 91a].

- Means for expressing both implicit and explicit synchronization must be provided. The implicit synchronization is based on the program counter and efficient to exploit instruction parallelism from a single thread, whereas the explicit synchronization names synchronization participants explicitly and is necessary to provide an efficient synchronization mechanism to exploit parallelism among multiple threads.

- Context switching must be fast. A multithreaded processor maintains multiple instruction streams (threads) and switches from one thread to another enabled thread based on certain events. Furthermore, such context switching occurs much more often in a multithreaded processor than in a conventional von Neumann processor. Therefore, the context switching is a primitive operation in any multithreaded processor and must be fast.

- Thread initialization and synchronization should be cheap and integrated into the instruction set. This means, thread initialization and synchronization should be performed directly with normal instructions instead of calling operating system routines. Further, because each pending synchronization requires a unique name, the synchronization namespace should be large. A small synchronization namespace may lead to additional overhead for managing it at run time.

- The main goal of the multithreading is to tolerate various latencies which lead to performance loss in conventional von Neumann machines. Therefore, the choice of thread size should take this consideration into account and avoid the effects caused by various *latency-sensitive operations*, such as remote loads or synchronizing loads due to access some shared objects, as much as possible.

Thus, a multithreaded execution model should combine the von Neumann with the data flow model and therefore take advantages of executing long sequential threads and of tolerating various latencies while providing efficient synchronization support for the parallel exection of a program. The remainder of this chapter is devoted to the definition of a multithreaded execution model and associated issues.

5.2 Basic Issues of the LDME

5.2.1 Program Representation

From the previous studies of data flow machines[Ian 90], it has become clear that a DATA FLOW PROGRAM GRAPH (DFPG) is a powerful and elegant representation for parallel programs because of its ability to express the partial ordering of instructions, its independence of architectures and its precise semantics. However, direct interpretation of a DFPG-derived graph as in a dataflow machine implies that all synchronization should be dynamic at the instruction level. It has been proved that this leads to poor performance on sequential code sections and to complex hardware requirements [Ian 90]. Multithreaded execution combines such sequential instruction sections into *large scheduling units*, i.e threads, thus, scheduling and synchronization occurs only at the thread level.

Before we go into details of the proposed multithreaded execution model, we first distinguish between two different types of latencies which should be hidden by multithreading.

> **Definition 5.1.** An instruction is said to contain a **static latency** when its operation latency can be statically determined at compile time.
>
> □

All instructions having a constant execution time such as simple arithmetic instructions or branch instructions contain static latency. On the contrary, there are some other instructions whose execution time is unpredictable at compile time, thus we have further

> **Definition 5.2:** An instruction is said to contain a **dynamic latency** when its operation latency can not be statically determined at compile time.
>
> □

In general, the remote load instructions and synchronizing loads contain dynamic latency, because their execution time depends mainly on factors such as network delay, network conflict, scheduling or computation time for other instructions. Moreover, some *complex instructions* such as integer/floating point multiplication, integer/floating point division, or some system instructions also contain dynamic latency, because the execution time of such instructions may vary from time to time.

Like other dataflow models, a multithreaded execution model should exploit the **locality** of program execution, i.e. cluster activities (threads) which are logically closely related into **code blocks**, which are scheduled and executed on one processing element. Hence, a program is represented by a collection of code blocks. Together with the definitions above, a code block in our multithreaded execution model is defined as follows:

> **Definition 5.3:** A **code block** is a digraph G $= (T, L, D, R)$, where $t_i \in T$ is a **thread** consisting of a sequence of instructions in which the result of an instruction with a dynamic latency must not be used by any instruction in the same thread, $d_{ij} = (i, j) \in D, i, j \in T, L$, implies the **dependency** of i on j, and $l_k \in L$ is a **latency synchronization thread** used to synchronize any latency sensitive operations, $r_i \in R$ is a **split transaction arc** that is either *directed towards* l_i (r_i

has either no source vertex or a source vertex $t_j \in T$) or *directed away* from $t_i \in T$ (r_i has no destination vertex). Further, the digraph G satisfies:

- G is acyclic, or it can be proved that no thread within G will be initiated more than once simultaneously.
- G is self-cleaning. When all results have been generated it must be the case that no threads within the code block will be executed afterwards, and
- synchronizing any threads containing dynamic latencies must be performed through latency synchronization threads in L.

□

The first requirement guarantees that code blocks are **reentrant**, i.e. the instruction text can be shared among any number of simultaneous invocations. This is similar to the *one-token-per-arc* rule in static dataflow machines. The self-cleaning requirement permits the invoker of a code block to reclaim the execution context after it receives all the results. The third requirement separates instructions with static latencies from instructions with dynamic latencies such as remote loads or synchronizing loads, etc. Within a code block, latency synchronization threads may be used to synchronize threads containing instructions with dynamic latencies, whereas between code blocks latency synchronization threads are used to build an *interface* to other code blocks or to remote memory blocks, which allows powerful synchronization and efficient hardware implementation. For the synchronization between code blocks or with remote memory, latency synchronization threads are similar to *inlets* in Culler's TAM execution model [Cul 91a]. Recognition of latency sensitive threads within a code block caused by any instructions except for accessing remote memory or remote processors is a major difference of our model from other proposed models and allows to achieve a more efficient implementation in multithreaded pipelines. We call our execution model **Latency-Directed Multithreaded Execution (LDME)**.

The structure of a code block may be as shown in Figure 5.1, where the dashed lines represent split transaction arcs, and R0 is the split request generated by an instruction with dynamic latency in the node T3. R1 and R2 are two remote access requests which are received by node L2 and L3, respectively. As seen, the nodes L1, L2 and L3 are three latency synchronization threads.

To simplify the understanding of our multithreaded execution model, in the following we will describe program execution by terms used frequently in the literature.

Each code block may be invoked several times during execution. Each invocation of a code block is called an **activation**. Associated with each activation is a set of memory locations

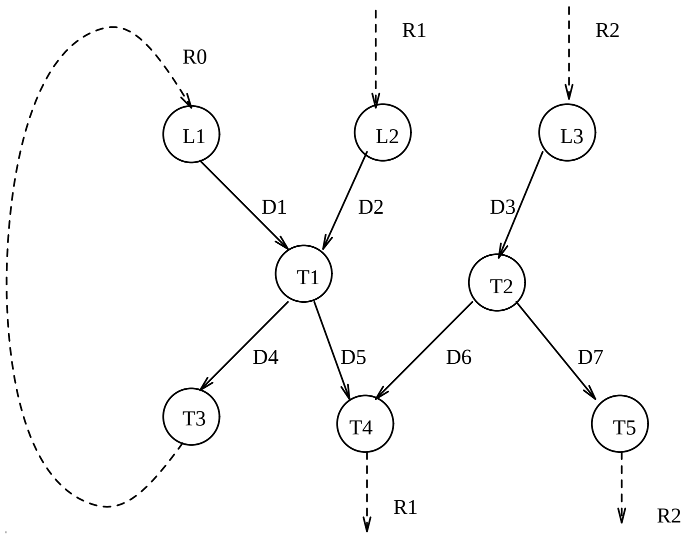


Figure 5.1. Structure of a LDME code block

used as local working environment called **frame** or **context** of the activation. An executing code block may invoke several code blocks concurrently, and the caller is not suspended as in a sequential language. Therefore, the set of frames at any time forms an **activation tree** rather than a stack. This parallel call scenario is often refered to as **fully parallel** execution [Pap 91a]. Further, to allow greater parallelism and to support languages with **non-strict** function call semantics, the arguments to a code-block may be delivered asynchronously; each will initiate a latency synchronization thread within the code block. An activation is said to be **enabled** whenever at least one thread in the activation is enabled. If an enabled activation is loaded into the processor for execution, we say that the activation is **resident**. Two activations may be resident on a processor[1]. A resident activation has access to all processor resource and remains executing until no enabled thread exists in the activation. Then, the activation is either **terminated** when all threads are terminated or **suspended** when there is at least one thread in the activation which is not enabled. A suspended activation may be unloaded from the processor and become enabled again when at least one remaining thread becomes enabled.

A thread becomes **enabled** if all data values for execution of the thread are available. Synchronization can be classified as **latency sensitive** and **non-latency sensitive** depending on whether a thread receives its operands from threads involving latency-sensitive instructions or from threads involving no latency sensitive instructions. Note that all external synchronizations due to remote loads, synchronizing loads or the resource manager are viewed as latency sensitive synchronization. For any non latency sensitive threads, synchronization is performed by detecting their **synchronization count** which records available operands. Only when

[1]This is only restricted by the current implementation which will be described in the next chapter. An extension to allow that more activations may be resident is possible by integrating more hardware contexts, which is an issue that will be pursued further.

the synchronization count becomes zero, the associated thread becomes enabled. An enabled thread is said to be **ready** when its associated activation becomes resident. A ready thread waits for the processor **resource** which includes a set of process registers such as program counter and the base registers. Once a thread acquires a free resource, it becomes **executing** or **active** and executes until completion.

Instructions within a thread are executed sequentially. Since latencies between instructions within a thread are predictable, for example latencies due to static data hazards or due to control hazards, the multithreaded pipeline can avoid all such inter-instruction latencies by interleaving multiple executing threads. The details are presented in the next chapter.

LDME provides four levels of synchronization: simple sequencing of instructions within a thread, scheduling of latency synchronization threads, scheduling of non latency synchronization threads and scheduling of enabled activations. As other hybrid von Neumann and data flow models, we let the compiler control thread-to-thread and activation-to-activation transitions as much as possible in order to use the least expensive form of synchronization. The compiler will produce all synchronization counts and latency synchronization threads explicitly.

5.2.2 Choice of the Thread Size

There are many strategies proposed for choosing the grain size, i.e for partitioning a program into smaller tasks. Most of these strategies combine a sequence of instructions into a large scheduling unit under the consideration of their logic relation and avoidance of dead locks. In his Large-Grain Dataflow model Dai chooses *basic block* [Dai 88] as his operation node, while Iannucci partitions code blocks into SQ's by clustering sequences of sequential instructions based on the fine grain dataflow program graph. Iannucci proposed some general issues of concern for optimal partitioning [Ian 90]:

- maximization of exploitable parallelism
- maximization of run length of tasks
- minimization of explicit synchronization
- deadlock avoidance
- maximization of machine utilization

It is not the focus of this work to develop the optimal partitioning strategy, but rather, to demonstrate the performance advantages of multithreaded architectures. For the LDME model we choose the partitioning technique which is *driven* by latency instructions in order to take advantages of multithreaded architectures.

> **Definition 5.4.:** A **latency free** thread is a sequence of sequential instructions in which no instruction uses the results produced by any instruction with dynamic latency within the same thread. □

As seen in definition 5.1. all threads (including latency synchronization threads) in the LDME are latency free. This choice is based on the following considerations:

- A thread constructed in this way will reduce the synchronization frequency, because once a thread is initiated, it continues executing until completion without any further inter-thread synchronization. Hence, the execution time of each enabled thread is deterministic.
- Static latencies within a thread (predictable operation latencies) due to data hazards or control hazards can be easily avoided by interleaving a small number of executing threads as discussed in 4.3.3. Thus, a small number of hardware contexts are required to achieve high pipeline utilization.
- Any instruction with dynamic latency can be *re-phrased* as split transaction and be synchronized later by latency synchronization threads. Hence, no complex hardware mechanism for synchronizing operations with undeterministic latencies is needed.
- Together with latency synchronization threads, latency free threads allow the compiler to take advantages of static information in a program and let the compiler control thread or activation transitions as much as possible in order to use the least expensive form of synchronization.
- More important, threads with deterministic execution time permit efficient resource utilization, which is actually the purpose of using multithreading, because unbounded execution time of threads may force unloading some suspended threads due to limited hardware contexts, for example, in the case when all executing threads are suspended. However, such suspension and resumption are typically expensive due to storing and restoring the associated contexts.

As mentioned before, a code block will be assigned to a single processing element (here a multithreaded uniprocessor) due to locality of processing as in most dataflow machines. This means that at any instant there is one or only a small number of threads executing. Therefore, fine grain thread size as in many other models becomes meaningless for threads belonging to the same activation. In contrast, fine grain threads lead to a large amount of synchronization between threads which are pure overhead. Hence, the LDME allows to exploit parallelism at some coarse level in order to achieve optimal overall performance. The compiler takes the responsibility for controlling the size of threads under the consideration of certain properties of a multithreaded architecture such as the length of the associated multithreaded pipeline.

In the following a simple example is used to demonstrate the synchronization principle of the LDME by using latency synchronization threads and to show the difference to other typical multithreaded execution models. A *rload-rload-subtract* $(A - B)$ computation may be realized on P-RISC [Nik 89] as follows:

```
    fork M1
L1: vA = rload pA    --   (A)
    jump N

M1: vB = rload pB    --   (B)
    jump N

N:  join 2 J         -- synchronization of responses
    C = vA - vB
```

The instruction *fork* initiates a new thread at label M1 to exploit fine grain parallelism. The parent thread continues at L1 and issues the remote load (A), which then suspends to await the response. The just forked thread executes at M1 and issues also the remote load (B), and then awaits the response. When the response arrives, the corresponding thread is resumed and completes the remote load instruction by storing the arriving value into vA or vB. *Join 2 J* is the synchronization instruction and will be executed twice, once by each thread. J is initialized to 0 and increased by each execution. If $J < 2$, the thread dies, otherwise it continues.

It is clear that two remote load instructions have dynamic latency, which blocks the further execution of threads. The suspended threads possess however all necessary processor resource for resumption of the thread. But this is an obvious **waste of processor resource**. With the LDME, we *split* all instructions with dynamic latencies into two parts: an initiate instruction and a latency synchronization threads like this as follows:

```
    fork M1
L1: rload pA:Sa       -- initiate remote load (A)

M1: rload pB:Sb       -- initiate remote load (B)

Sa: vA = store (V)    -- store incoming value (A)
    start N           -- synchronizing N
```

```
    Sb: vB = store (V)     -- store incoming value (B)
        start N            -- synchronizing N

  N(2): C = vA - vB
```

The instruction *rload pA:Sa* initiates the remote load on the address pA, and the response will initiate the latency synchronization thread Sa. When the response (say A) arrives, the latency synchronization thread Sa is initiated, which stores the incoming value into vA. The instruction *start N* then tries to synchronize another thread Sb, where the number 2 in N(2) is the *synchronization count*, i.e. the number of required incoming data values to initiate the thread N. If the two instructions *start N* of both Sa and Sb are completed, then the thread N is initiated.

The difference of these two execution models is obvious. Comparing the two programs above alone, two main features of the LDME programs can be drawn:

- All threads are *non-blocking*, i.e. there is no suspension for any thread. Once a thread is initiated, it continues executing until completion. Only threads in execution possess the processor resource.
- All instructions with dynamic latencies are rephrased as split phase transactions, the associated *continuation*, i.e. a completion specification of the thread awaiting for response, contains only the identity (thread number) of its latency synchronization thread instead of the complete context of the suspended thread like the execution model on P-RISC. This can reduce the amount of information carried on message to the network.

5.2.3 Construction of LDME programs

The construction of a directed graph for a code block consists of two steps: partitioning a code block into threads (T and L) and connecting the resulting threads with directed arcs (D and R). The problem of exploiting a reasonable parallelism within a code block is more complicated and can only be solved through data flow analysis under the consideration of certain architecture properties. In the following, we will demonstrate the construction of LDME programs through a typical example DAXPY, which is also used to show the multithreaded model of *T machine[Nik 92]. The reader is encouraged to compare the following programs with the programs for *T.

An example: DAXPY

DAXPY is the inner loop of the Linpack benchmark [Nik 92]

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

We assume that the basic associated execution context (frame) contains the following data, which is accessed relative to its base-frame-pointer BFP:

...
N: Loop constant
i: Loop variable
XP: Pointer to X[0]
YP: Pointer to Y[0]
A: constant
Mtmp: temporary variable
...

Further, we assume that the execution time of multiplication is multiple cycles, therefore, it is reasonable to view multiplication as an operation with dynamic latency in a multithreaded pipeline with deterministic pipeline stages.

1. Uniprocessor code

For a uniprocessor, the unique issue of concern for constructing a LDME program is based on instructions with dynamic latencies. Note that there is only local memory in a uniprocessor, which contains the arrays X and Y, hence, no memory latency will occur. The uniprocessor code for DAXPY is shown in Figure 5.2, where the names with "R" are the general purpose registers of the allocated register set[2] , the names with "( )" the memory locations in a local or remote memory, the solid lines represent dependent arcs, and the dashed lines split transaction arcs.

As shown, there are four threads in the code above. Note that the multiplication instruction *RXI*Ra* is split. *Synt* is the corresponding latency synchronization thread which is initiated automatically when the multiplication completes. *Synt* stores the result into the frame and then initiates another thread *A*1 for further computing. *stop* explicitly terminates a thread, otherwise a thread is implicitly terminated after the execution of its last instruction. Note that the crossed circle represents the *merge* relation of two input arcs.

Here we see another accompanying consequence of the LDME, which can degrade the performance, i.e. each thread must perform many loads and stores. The reason of using such loads and stores is that there is no implicit saving or restoring of registers which are released after termination. Thus, it is forced to load and store the values in registers repeatedly. Moreover, the thread *A1* is directly initiated by the latency synchronization thread *Synt* and does

[2]Assuming that a multithreaded processor contains multiple register sets in order to maintain multiple executing threads, which are necessary for masking possible latencies due to hazards.

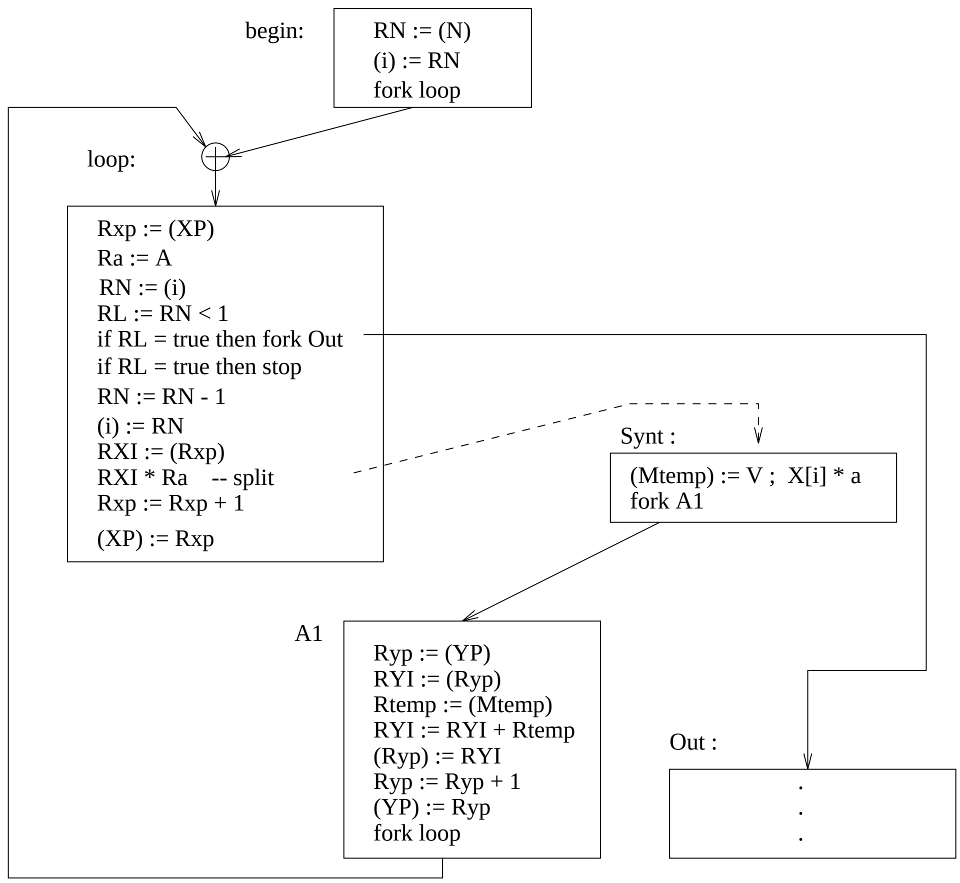


Figure 5.2. The directed graph of the uniprocessor DAXPY code

not require any synchronization with other threads, hence, two threads *Synt* and *A1* can be **merged** together to avoid some superfluous instructions.

To solve the first problem we define some *global* registers for each activation, which allow sharing of data across threads. Global registers will be released only when the *resident* activation is terminated or unloaded from the processor. Thus, global registers for each activation allow the compiler to control the use of such registers in order to reduce loads and stores across threads. Now, the improved uniprocessor assembly code can be rewritten below:

```
      load Gxp, XP        -- load pointer to X
      load Gyp, YP        -- load pointer to Y
      load Ga,  A         -- load constant a
      load GN,  N         -- load loop constant
      ltp  RL, GN, 1      -- if GN < 1
      tfork RL, Out       -- zero-trip loop if less
      tjump RL, End       -- stop if less
      fork Loop           -- initiate Loop
 End: stop                -- stop

Loop: load RXI, Gxp       -- X[i] into RXI
      fmul (Ra, RXI):Synt -- split multiplication
```

```
      add Gxp, Gxp, 1       -- increase Gxp

Synt: mload Rtmp, V         -- load result of mult. into Rtmp
      load RYI, Gyp         -- Y[i] into RYI
      add Rtmp, Rtmp,RYI    -- a*X[i]+Y[i]
      store Gyp, Rtmp       -- store Y[i]
      add Gyp, Gyp, 1       -- increase Gyp
      sub GN, GN, 1         -- decrease N
      gtp  RL, GN, 0        -- if CN >= 1
      tfork RL, Loop        -- initiate Loop

Out:  ...
```

Note that the names with "G" are global registers of the resident activation. As seen, the loop consists of two threads now. Different from the methods for resolving dynamic latency in chapter 3 (scoreboard, reservation station etc.), multithreading is used here by splitting instructions with dynamic latencies, and an associated latency synchronization thread will be activated automatically when corresponding operations are finished.

2. Multiprocessor code

For multiprocessor code we have to consider another issue of concern in constructing LDME programs, i.e. latencies due to remote accesses which may be either remote loads or remote synchronizing loads. Because such latencies are often long and unpredictable, more threads are required to mask such latencies.

Suppose X[I] and Y[I] are stored on remote memories (or nodes) in a multiprocessor. Therefore, all loads of fetching array elements have to be split, thus, the processor can be freed to do other work during accesses. Once a response arrives at the processor, an associated latency synchronization will be initiated, which tries to join (synchronize) with the other responses using a synchronization count. If the synchronization succeeds, the thread for using the data is initiated. After execution, a *rstore* instruction stores the result in the remote memory. Meanwhile, the *rstore* acknowledges all arriving data at the processor, which is introduced to guarantee *self-clearing* of programs just like in other dataflow models.

The multiprocessor code is also organized in a single code block, and its extended frame layout is shown below:

...
N: Loop constant
XP: Pointer to X[0]
YP: Pointer to Y[0]
Xa: Copy of a*X[i]
YI: Copy of Y[i]
A: constant
...

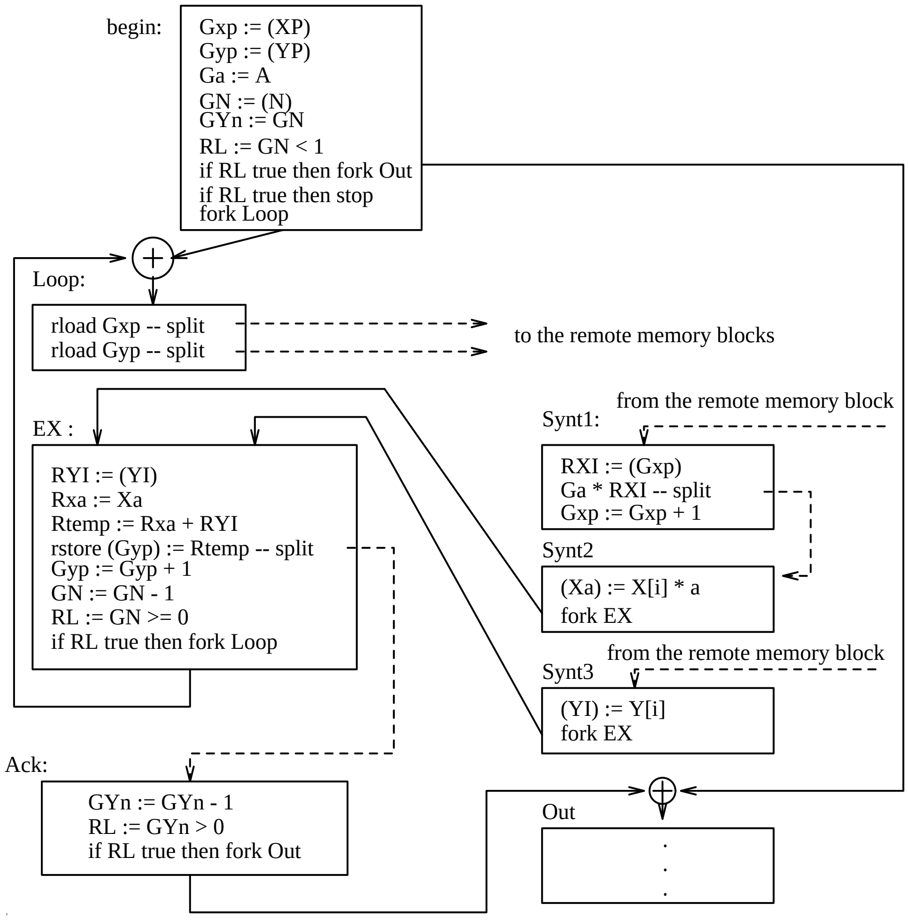


Figure 5.3. The directed graph of the multiprocessor DAXPY code

Figure 5.3 shows the directed graph of the code block. The corresponding assembly code is given below.

```
        load Gxp, XP          -- load pointer to X
        load Gyp, YP          -- load pointer to Y
        load Ga,  A           -- load constant a
        load GYn, N           -- load loop trip used for ack.
        load GN,  N           -- load loop trip
        ltp  RL, GN, 1        -- if GN < 1
        tfork RL, Out         -- zero-trip loop if less
        tjump RL, End         -- stop if less
        fork Loop             -- initiate Loop
  End:  stop                  -- terminate

 Loop:  rload Gxp:Synt1       -- initiate load X[i]
        rload Gyp:Synt3       -- initiate load Y[i]

Synt1:  mload RXI, V          -- load the arriving X[i] into RXI
        fmul (Ra, RXI):Synt2  -- split multiplication
        add Gxp, Gxp, 1       -- increase Gxp

Synt2:  store Xa, V           -- store the arriving a*X[i] into Xa
        fork EX               -- synchronizing with Y[i]
```

```
Synt3: store YI, V          -- store the arriving Y[i] into YI
       fork EX              -- synchronizing with a*X[i]


  Ack: sub GYn, GYn, 1      -- record the completed rstores
       eqp RL, GYn, 0       -- if GYn = 0
       tfork RL, Out        -- initiate Out if true

   EX: load RYI, YI         -- load copy of Y[i]
       load Rxa, Xa         -- load copy of a*X[i]
       add Rtmp, RYI, RXa   -- a*X[i] + Y[i]
       rstore Gyp:Ack       -- rstore the result
       add Gyp, Gyp, 1      -- increase Gyp
       sub GN, GN, 1        -- decrease N
       gtp RL, GN, 0        -- if CN >= 1
       tfork RL, Loop       -- if true then initiate Loop

  Out: ...
```

Synt1, Synt2, Synt3 and *Ack* are four latency synchronization threads used to synchronize the remote accesses and multiplication. As seen, the code above is constructed under the consideration of splitting all instructions with dynamic latencies. If enough parallelism is available, the code may consist of many small threads which are asynchronously initiated and synchronized. Because of cheap thread initiation and thread synchronization, such parallel threads will be used to mask all dynamic latencies.

5.2.4 Some Characteristics of the LDME Model

The above example shows the following main characteristics of the LDME, which are of importance for its efficient implementation:

- Instead of complex hardware synchronization mechanism such as frame memory with *presence-bits*, latency synchronization threads are used to synchronize split operations and to schedule threads. Latency synchronization threads are usually short and should be a small fraction in comparison with normal sequential threads.
- Split phase transaction is the key technique to mask all possible unpredicatable latencies. This also allows to avoid complex hardware mechanism to resolve the hazards due to unpredictable computation time such as multiplication in the above example.
- By separating normal sequential threads from latency synchronization threads, the proposed approach permits efficient hardware implemention, because latency synchronization threads are usually short and can be executed on a special coprocessor, while all

normal threads are executed on a multithreaded processor which is optimized for execution of sequential threads and mask inter-instruction hazards by interleaving a small number of threads.

- By accessing the same frame, no actual data flows between threads take place. By introducing some *global registers* for each resident activation, this approach allows more compiler interference by managing registers in order to reduce the number of used loads and stores.

5.3 Implementation Issues

So far, the LDME should be viewed as a *logical* model for a multithreaded processor (uni- or multiprocessor). In this section, we deal with some details of implementation issues of the LDME. Some important aspects of implementation will be presented, which include representation of threads, storage hierarchy, synchronization and scheduling etc. The implementation presented here has been directly influenced by von der Heide's PUMA [VdH 86] and the author's previous multithreaded machine MTMA [Fan 91][Fan 92a][Fan 92b]. These implementation issues have also been coordinated with its actual hardware architecture support. A multithreaded processor supporting the LDME will be presented in detail in the next chapter.

5.3.1 Thread

A thread $t \in T$ of a code block G (see definition 5.3) is characterized by a triple consisting of

$$< \mathrm{IP}, Lg, Sc >$$

where *IP* is the instruction pointer, Lg the length of the thread and Sc the synchronization count. Each thread is a sequence of sequential instructions and contains no loop, therefore, Lg is also equal to the number of executed instructions within the thread. A loop will be treated as a code block (see the next section). We integrate the *synchronization count* into the above triple only due to hardware implementation considerations. Logically, the synchronization count Sc is equal to the entry count in [Nik 89]. A thread with $Sc > 0$ is called a *synchronizing thread* which will be initiated by more than one other thread, whereas a thread with $Sc = 0$ requires no synchronization and is therefore called *nonsynchronizing thread*. Because all three terms in the triple can be determined statically, the above triple is also called *static thread descriptor*.

As mentioned, a thread is executed only when its corresponding activation is resident. At the beginning all static thread descriptors reside in a special high speed memory called *descriptor memory*. Static thread descriptors are modified during program execution. A static thread descriptor will be extracted from the descriptor memory and stored into a thread queue when the corresponding thread becomes enabled.

An enabled thread becomes executing when it gets a free resource. A thread is executed under the environment of its corresponding activation and is then characterized by

$$< Fp, Dp, Rg, \mathrm{IP}, Lg >$$

where Fp is the data frame pointer of its activation, Dp the descriptor frame pointer and Rg the register set pointer. Likewise, the above tuple is called *dynamic thread descriptor* which is constructed at run time.

The choice of the above representation of a thread is based on the following considerations:

- All terms in the static thread descriptors can be determined at compile time. It is not necessary to generate thread descriptors at run time, which may require more hardware support.

- Saving all static thread descriptors in a seperate memory (descriptor memory) allows to perform synchronization and initiation of a thread with a single machine instruction in the actual implementation (see the next chapter).

- As mentioned, only two activations are resident (i.e. executing) at any time. However, several other enabled activations may exist in the processor. It is possible that an enabled activation may accumulate several enabled threads (for example, some arguments are supplied), i.e. their thread descriptors must be saved in the corresponding thread queue. Again, separating the descriptor memory from the data frame allows writing such thread descriptors in their thread queues with a single instruction.

- The descriptor memory is much smaller than the instruction memory, which allows referencing threads with short address and leads to efficient encoding of instructions.

When a code block is invoked, a descriptor frame must be allocated for the activation, which also includes initiation of the allocated descriptor frame. The overhead here is thus to copy all static thread descriptors into the descriptor frame. Similar overhead also occurs in using entry counts in the data frame and involves initiating all entry counts before any execution[Cul 91a].

However, all threads $l_i \in L$ (i.e. latency synchronization threads) are treated differently. Like other execution models [Nik 92], the thread descriptors of latency synchronization threads are formed when arriving responses of split operations become available. The reason for different treatment of latency synchronization threads is given below.

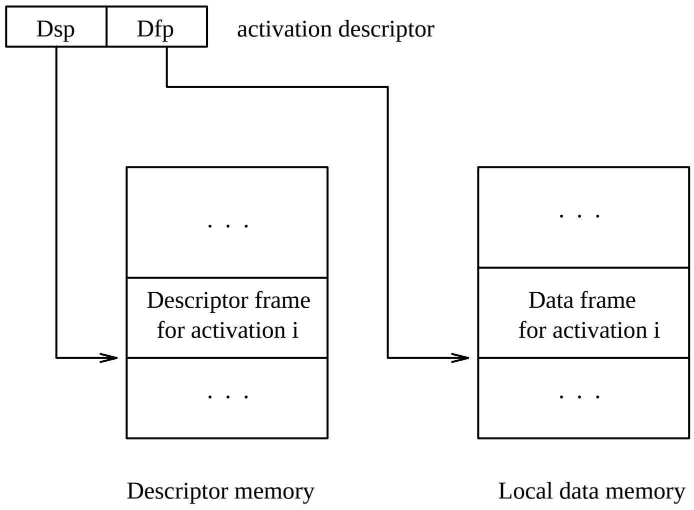


Figure 5.4. An activation environment

5.3.2 Activation

Like in other dataflow and multithreaded machines [Arv 86][Cul 91a], we call each invocation of a code block an activation. Because an executing code block may invoke several code blocks concurrently, it is necessary that each activation has its own executing environment, which is allocated by the run time manager at the time of its invocation[3]. For the LDME model such an activation environment consists of a descriptor frame and a data frame and is referenced by an *activation descriptor* which consists of the pointer to the descriptor frame and the pointer to the data frame $< Dsp, Dfp >$, as shown in Figure 5.4.

The descriptor frame contains a queue descriptor (QD) and all static thread descriptors (TDs) as shown in Figure 5.5. The queue descriptor references to the thread queue holding thread descriptors of enabled threads in the data frame.

queue descriptor
thread descriptor 1
thread descriptor 2
...
thread descriptor n

Figure 5.5 Structure of a descriptor frame

The data frame is shown in Figure 5.6. *Argument slots* contain all input arguments from its parent activation and results from its child activations. An activation descriptor and a synchronization thread pointer for its parent activation and several activation descriptors and

[3] It should be noted that the dynamic resource allocation for each activation has been researched intensively by dataflow researchers and almost become the standard technique used in many dataflow or multithreaded machines. For the LDME model this technique is also necessary to exploit sufficient parallelism in a program in order to hide various latencies.

synchronization thread pointers for its child activations are generated by the *resource manager* when the code block is invoked. They are used to transfer arguments and results between caller and callee and to reference latency synchronization threads in a separate synchronization processing unit (see the next chapter). Local slots are the working space for the activation. At last, the thread queue slots are used to save thread descriptors of all enabled threads. The size of the thread queue will not overflow at any time, because the maximum number of threads within an activation can be determined statically at compile time.

activation descriptor for parent
synchronization thread pointer for parent
activation descriptor for child 1
synchronization thread pointer for child 1
...
activation descriptor for child m
synchronization thread pointer for child m
argument 1
...
argument n
local slot 1
...
local slot k
thread queue element 1
...
thread queue element s

Figure 5.6. Structure of a data frame

Note that both descriptor frame and data frame are generated statically by the compiler. By invoking a code block, a descriptor frame and a data frame are allocated. Further, initiation of the descriptor frame and the data frame must also be performed by the resource manager prior to executing the activation. The relation between descriptor frame, data frame and program code of code blocks as well as the basic structure of storage resource are depicted in Figure 5.7.

The following features of the storage structure of the LDME deserve some more comments:

- No presence-bits in each location of the data memory and no *synchronization counts* for synchronization are required, because synchronization between threads is performed through the synchronization count in static thread descriptor, which simplifies the implementation of local memory.

- Because the length of a thread is statically determined at compile time, no explicit instruction *stop* is required. The hardware checks the length of threads and starts executing another enabled thread whenever the length of the executing thread has decreased to zero.

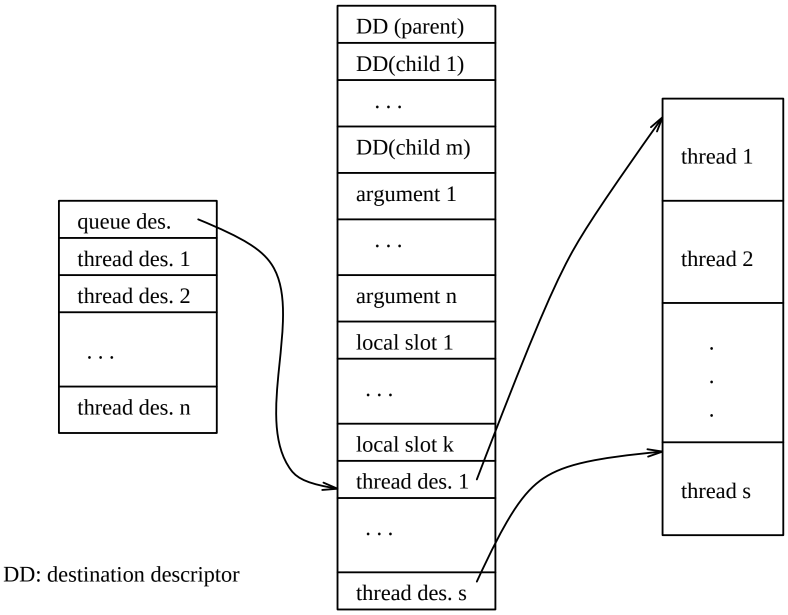


Figure 5.7. Basic storage structure of the LDME

- Data structures are not allocated directly in the local data frame, but in a seperate heap. The data frame only holds their descriptors (see the DAXPY example in the last section). An efficient access to data structures through descriptors is performed by split phase transaction. We assume here that the heap is equipped with I-structure semantics [Arv 86].

- We treat all latency synchronization threads differently from the non-latency synchronization threads. Latency synchronization threads are initiated by responses of split operations and are usually small. Moreover, latency synchronization threads belonging to different activations (resident and nonresident) may be initiated simultaneously, for example, in transfering arguments to callees or returning results to callers. In this case, latency synchronization threads belonging to a nonresident activation must also be executed without suspension, wheareas non latency synchronization threads can be scheduled for execution only when their corresponding activation becomes resident.

- The initiation and termination of activations are done dynamically. Therefore, logically all enabled activations form an *activation tree* instead of an activation stack as in sequential execution. This activation tree may be mapped to multiple processing elements and is executed in parallel.

5.3.3 Synchronization and Scheduling

In the LDME there are four levels of synchronization during program execution: simple sequencing of instructions within a thread through IP, scheduling of latency synchronization

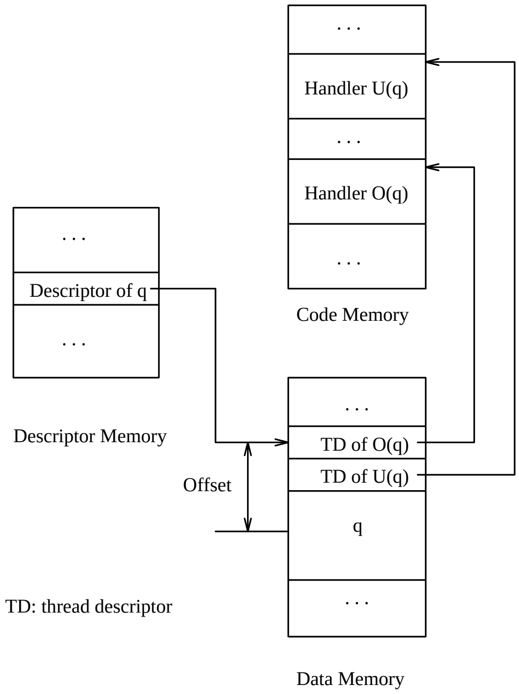


Figure 5.8. Extended queue structure

threads through responses of split latency sensitive operations, scheduling of non-latency synchronization threads through explicit instructions and scheduling of enabled activations or resumed activations. We proposed queue-based primitives to perform the two latter scheduling [Fan 91], i.e. scheduling of non-latency synchronization threads and scheduling of activations. The queue-based primitives are based directly on buffer-objects and the associated event-driven control mechanism in PUMA [VdH 86].

Queue-structure

Usually, a queue is a linear structure for which access is performed in FIFO manner. Here, the conventional queue concept has been extended: to each queue two exception handlers are assigned, which can be programmed freely. These two handlers are used to handle two exceptional events: **QueueOverflow** and **QueueUnderflow**. So a complete queue is represented by

$$< q, O(q), U(q) >$$

where q is a circular array, *O(q)* its overflow handler and *U(q)* its underflow handler. These two handlers can also be viewed as two threads, whose thread descriptors as well as queue elements are constructed together as a data structure stored in the data memory. A queue is accessed through its queue descriptor stored in the descriptor memory. Accesses to the descriptor memory are atomic and involve reading the queue descriptor, modifying it and storing it back into the descriptor memory.

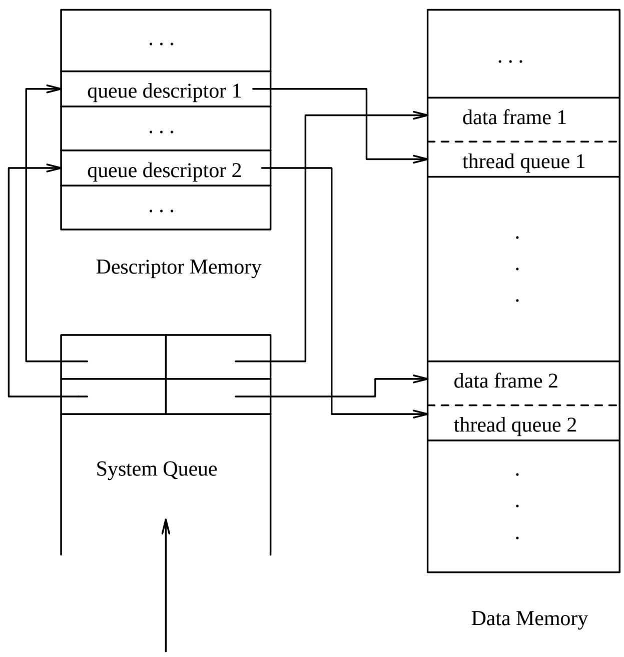


Figure 5.9. Two level scheduling

A mechanism called *event-driven control* is used to perform the following operations whenever an exceptional event is detected:

- The thread descriptor of the executing thread is stored in its context temporarily, i.e. suspending the executing thread. All instructions belonging to the suspended thread in the multithreaded pipeline are eliminated, i.e. the pipeline must be cleaned up after exception.
- One of the thread descriptors (*O(q) or U(q)*) is loaded from the queue structure and activated immediately.

Thus, the event driven control mechanism will trigger the corresponding handler if an exceptional event occurs during queue access. The handler performs the desirable action.

Two level scheduling

As noted previously, only two activations are scheduled for execution at any time, while other enabled activations must wait for free resources. Using queue-based primitives, a two level scheduling can be implemented efficiently (Figure 5.9).

At the first level each activation has its own thread queue where the descriptors of enabled threads are stored. For nonresident activation the descriptor of an enabled thread is inserted into its thread queue when the thread becomes enabled. This is performed by a latency synchronization thread. For resident activation the descriptor of an enabled thread will be inserted into its thread queue only when all processor resources have been allocated. Otherwise this enabled thread gets a processor resource and starts execution immediately, and its descriptor will then not be inserted into the thread queue. During execution the next thread descriptor will automatically be read from the thread queue of one of the resident activation when one executing thread terminates. In our actual implementation, we distinguish the exceptional events of access to thread queues for resident activation and for nonresident activations. For resident activation, some threads are being executed, while other enabled threads which have not got processor resources for execution remain in their thread queue. The access to the thread queue of resident activation is implicitly performed by hardware, i.e. a thread descriptor is read from its queue whenever a processor resource for executing another enabled thread is available. Therefore, an exceptional event *QueueUnderflow* occurs when the thread queue is empty and no threads are in execution, i.e. all enabled threads of this activation are terminated. For a nonresident activation, all enabled threads reside in their thread queue, which is accessed (read) explicitly by queue read instructions of the resource manager when this activation becomes resident. If no descriptor exists in the queue (i.e. there is no enabled thread in the activation), the handler U(q) switches the activation and inserts its activation descriptor into the system queue again, otherwise the activation becomes resident.

At the higher level another queue called *system queue* is used to save activation descriptors of enabled activations or suspended activations. An activation descriptor is read from the system queue whenever the resident activation is terminated or suspended. If the system queue is empty or full, its associated handler delivers the information to the global resource manager whether the processing element is *idle* or *overloaded*, which should help load balancing in task distribution.

In the current LDME model the queue-based mechanism is used only to schedule threads and activations. However, the queue is a useful data structure for many algorithms. In PUMA, the queue is one of the primitive objects supported by the hardware and has shown its advantages and its flexibility in constructing the execution pipeline, the storage management routines [Vogt 90] etc. In a previous multithreaded machine proposed by the author, queue-based primitives were used to perform communication and synchronization, to implement several important operations which are frequently used in parallel programming etc. [Fan 91].

5.4 Parallelism in the LDME

Four scheduling levels supported by the LDME allow to exploit parallelism at different levels. It is worth reviewing how the various forms of parallelism can be exploited with corresponding architectural support:

- *Intra-thread* parallelism is the basic form of parallelism for keeping the pipeline full, as in a conventional pipelined processor. Unlike conventional pipelined processors, no complex mechanisms for detecting and resolving various forms of pipeline hazards are employed. Instead, all such hazards and operation latencies are masked by parallelism between threads.

- *Inter-thread* parallelism within a code block is the most important form of parallelism for hiding various latencies, due to pipeline hazards, due to long operation latencies and due to long remote access latencies which degrade performance of multiprocessors based on von Neumann machines. Fast context switching between threads is required to mask these low level latencies. The use of multiple register sets, one for each executing thread, keeps the cost of context switching very low.

- *Inner-loop* parallelism is supported by representing each loop as a code block and providing efficient means for sending data between codeblocks. This requires fast allocation and deallocation of contexts and fast argument/result transfer. Moreover, latency synchronization threads can be used to support *nonstrict* invocations, i.e. invoke a code block before all its arguments are generated. At the same level lies the procedure call parallelism which can be exploited with the same mechanism.

- *Outer-loop* inter-iteration parallelism is supported by representing the storage for an iteration explicitly, and allowing multiple storage areas to be named and addressed as in [Ian 90]. It is also possible to unravel outer-loop at run time by executing some primitives like **init_loop** and **init_iter** in [Dai 88].

The question related to the exploitation of loop parallelism and procedure call parallelism are beyond the scope of the present work. It is claimed that solutions applicable to machines such as Iannucci's hybrid machine [Ian 90] or Dai's Large-Grain-Dataflow Machine [Dai 88] are also equally applicable here, because of the following relevant similarities:

- A program is organized as a collection of code blocks which can be initiated and terminated dynamically. Demand for context-specific storage is not only bounded, but also is known *a priori*. Each code block carries with it a record of descriptor frame and a record of data frame which are determined at compile time.

- *Nonstrict* invocation of loops and procedures, which allows to exploit parallelism efficiently.

- Similar data frame structure of invoked code block (activation), which supports fast argument/result transfer between activations. Data structure will not be allocated in a local data frame, but in a separate heap which possesses I-structure-like semantics to support parallelism and synchronization at data element level.

Likewise, we claim that solutions to handle finite resources while exploiting parallelism for other machines are also applicable here.

5.5 Summary and Remark

In dataflow computation, synchronization occurs on each instruction, and each synchronization event requires a unique name, therefore, a large synchronization name space is necessary, which is restricted actually by the size of the waiting matching unit. If we form a sequence of sequential instructions as a large scheduling unit (thread), synchronization occurs thus only between threads. Therefore, the associated synchronization name space is dramatically reduced. In the LDME model, we integrate the synchronization name space into thread descriptors by recognizing that synchronization events and basic thread characterization can be statically determined at compile time and integrating synchronization information into thread descriptors allows fast context switching (thread switching) in hardware implementation (see the next chapter), because no access to the activation frame (data frame) must be performed before thread scheduling. This is important for a multithreaded pipeline where access to the data memory is performed at a late stage.

The LDME model is based on various related research such as Iannucci's hybrid machine [Ian 90], Culler's TAM [Cul 91a], Arvind and Nikhil's P-RISC [Nik 89] and *T [Nik 92], and distinguished from them not only by providing mechanism to support tolerance of latencies and synchronization, but also by trying to improve the performance of multithreaded pipeline, which has been payed little attention in multithreaded research. In the next chapter, we will present a design of such a multithreaded pipeline.

Chapter 6

Design of a LDME Processor

So far the latency-directed multithreaded execution model (LDME) has been described. As mentioned in the last chapter, the LDME takes into account some potential hardware implementation advantages. For example the *synchronization count* for synchronization threads are directly integrated into their thread descriptors instead of reserving a location in their local data frame as in some other dataflow or multithreaded processors [Pap 91a][Ian 90]. In this chapter, we will present a concrete design of a LDME processor, which includes the detailed implementation considerations of the overall machine organization, the processing element, the instruction set architecture and the detailed design of a multithreaded pipeline.

6.1 Overview of a LDME Processor

The LDME is a model of MIMD computation. A LDME processor should be applied either as a uniprocessor or as a multiprocessor which comprises a collection of processing elements (PE's), connected to each other via a suitable interconnection network and to a set of memory modules which are constructed together as a global data structure storage containing I-structure semantics. The overall machine organization of a LDME multiprocessor can be sketched conceptually as in Figure 6.1.

I-structure storage has been discussed manifoldly and applied in many dataflow machines [Arv 86][Arv 90]. Hence, its structure will not be reiterated here. It is sufficient to assume that the storage units collectively implement a single global address space. Each unit accepts requests to *remote* LOADs and STOREs, performs synchronization and responses in a pipelined fashion. Communication between processing elements or accesses to the I-structure storage are performed through *split phase transaction*, and the corresponding responses can return in any order.

The network's internal structure is of little concern here, because the differences of networks

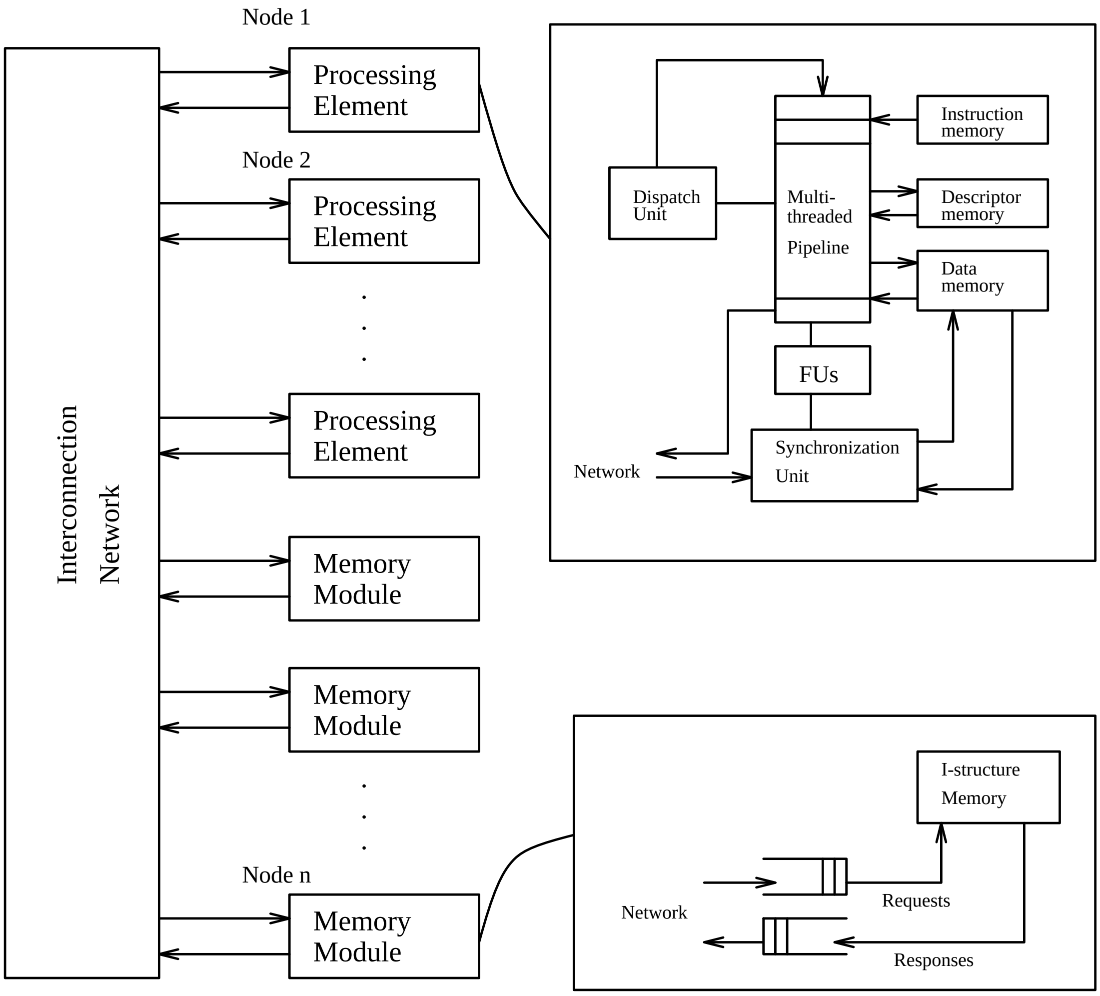


Figure 6.1. The overall structure of a LDME multiprocessor

can be reasonably distinguished by their different *transit latency* through the network. For example, a multistage packet switch network has a transit latency of $log(n)$, where n is the number of inputs. Further, we assume that messages can be accepted in a pipelined fashion, and that the acceptance/delivery rate matches both the processor and the structure memory.

A fundamental issue in a multiprocessor concerns the mapping of a computation onto a set of physical processing elements. For a LDME multiprocessor, this corresponds to the mapping of codes, descriptor frames and data frames across processing elements. To take advantage of locality, we employ the strategy in which the code, the descriptor frame and the data frame, which belong to the same activation, are allocated to a single processing element. This strategy has also been employed in many dataflow machines such as MIT's TTDA and Papadopoulos's Monsoon [Pap 91a].

By insisting that frame references are local to a processing element, we do not support load balancing algorithms that move activations, once allocated, among processors. Instead, load balancing could be considered when a new activation is initiated. In the previous chapter, we proposed a two level scheduling strategy. A system queue holds all enabled activations in the associated processing element. A just initiated activation can be allocated under the consideration of load situation of such system queues across processing elements. There are many strategies proposed for other multiprocessor systems. Grafe and Hoch proposed an interesting multistage packet switched network called *load balancing network* for their EPSILON-2 dataflow multiprocessor [Gra 91]: this extended multistage packet switched network is also responsible for load balancing by distributing activation frame requests across processing elements. The basic idea of the load balancing network is to route the activation frame requests intelligently to the least loaded processing element in the system and then to allocate an activation frame there. The routing of the activation frame requests through the network is guided by load indicators at network switching elements. Load indicators are propagated back from the processing elements and reflect the load state of each processing element.

Since the activation frame and the corresponding code are local to a processing element, we are assured that instructions which perform frame accesses are non-blocking. In contrast, an instruction accessing an arbitray storage location (say reading a frame local in a remote processing element or in the shared structure memory) is split, and the operation is completed by its latency synchronization thread when the response arrives. Thus, no instructions may be suspended during program execution, which is the key difference of our model from many other dataflow or multithreaded machines where synchronizing loads for frame locations may block instructions to complete [Nik 89][Nik 92].

6.2 Architecture

In this section, we present the architecture considerations of our preliminary multitihreaded processor, which include the supported hardware types, instruction set, and storage organization. As mentioned, we will not deal with multiprocessor issues such as network structure or I-structure memory, but concentrate on the architecture of the processing element.

6.2.1 Hardware Types

The processor has a fixed word size of 36 bit – 32 bits as data/pointer and 4 bits as type tag.

type	data/pointer
4	32

The choice of a 32-bit data width for this preliminary LDME processor is based on the observations of many modern RISC-processors. The basic LDME processor architecture is extended from the well-known RISC-processor DLX [Hen 90] by integrating some hardware mechanisms into the pipeline, which are required to support multithreaded execution. Like other dataflow machines such as Monsoon [Pap 91a] and EPSILON-2 [Gra 91], the processor has to handle several different data types such as descriptor frame pointers, data frame pointers, etc. Integrating data type information permits dynamic check of data types during instruction execution. In the following data types the field *type* is omitted.

Integer Number

Integer numbers are 32 bit, unsigned or signed two's complement notation. Integers will also be used to represent **boolean** values. We assume that the processor recognizes a **false** value as a word of all zero and a **true** value as anything else. Boolean values are useful to express results of comparison operations.

Floating Point Number

Floating point numbers conform to the IEEE Standard 754 for double precision as follows:

Sign	Exponent	Mantissa
1	11	52

Single precision floating point numbers are provided as well and represented as follows:

Sign	Exponent	Mantissa
1	8	23

There are special instructions for floating point operations. To simplify the implementation of the LDME architecture, all floating point instructions define only single precision floating point operations in this design. All double precision operations will be performed through corresponding software routines.

Thread Descriptor (TD)

A thread descriptor contains three terms which are statically determined at compile time:

IP	Lg	Sc
20	8	4

where IP is the instruction pointer referencing to the local instruction memory, Lg the length of thread, and Sc the synchronization count. Note that the compiler generates thread descriptors only for threads ($\forall t_i \in T$), i.e. normal threads. Latency synchronization threads are treated differently.

Queue Descriptor (QD)

A queue descriptor is a pointer to the thread queue of an activation, which is allocated within the data frame. Its format is shown as follows:

–	Head	Tail	Length
2	10	10	10

As shown, the first two bits are not used, $Head$ points to the head of the queue, $Tail$ points to the tail of the queue, $Length$ is the length of the queue. For each activation, the compiler allocates sufficient area for saving thread descriptors of all possibly enabled threads. Therefore, no exceptional event **QueueOverflow** will occur. $Head$ and $Tail$ are used for determining the exceptional event **QueueUnderflow**.

Note that a queue occupies a continous area within a data frame. The base address of the queue area is computed based on the data frame pointer. Additionally, a queue structure can also be used for other purposes. In this case, extra queue area must be allocated and accessed through an explicit base address. This issue, however, is not included in this study.

Data Frame Pointer (Dfp)

A data frame pointer contains the base address of a data frame allocated in the local memory. Data frame pointers are 20 bit wide, the upper 12 bits of a word are not used.

Descriptor Frame Pointer (Dsp)

A descriptor frame pointer contains the base address of a descriptor frame allocated in the local descriptor memory. Descriptor frame pointers are 12 bit wide, the upper 20 bits of a word are not used.

Activation Descriptor (AD)

An activation descriptor consists of a data frame pointer Dfp and a descriptor frame pointer Dsp as follows:

Dsp	Dfp
12	20

An activation descriptor is only allocated when an activation is initiated, then the activation descriptor is first loaded into the system queue before the corresponding activation becomes resident. Activation descriptors are also used to build request messages for accessing remote processing elements. Note that no explict activation descriptors are required for split operations within the same processing element and for accesses to the I-structure memory. These cases, the activation descriptors will be implicitly extracted from the current execution environment.

I-structure descriptor (ISD)

An I-structure descriptor encodes an address in a I-structure memory. I-structure descriptors are represented as the *segmented address* NODE:OFFSET as follows:

NODE	OFFSET
12	20

where OFFSET is a pointer to a location in the memory module NODE.

Code Block Pointer (CBP)

A code block pointer points to a code block in program memory. Because code blocks do not span across processing elements, a code block pointer is represented as follows:

NODE	Code Block Address (CBA)
12	20

where NODE is the processing element number, and CBA the address of a code block in program memory. Code block pointers are used by the run time manager to invoke activations.

Synchronization thread pointer (STP)

All latency synchronization threads of a code block are treated differently and referenced by a pointer, which is represented as follows:

NODE	Instruction Base Address (IBA)
12	20

where NODE is the processing element number, and IBA the base address of the code of latency synchronization threads, which are allocated in a continous memory segment in the synchronization unit (see 6.3.2).

6.2.2 Implicit Hardware Types

In contrast to the data types mentioned above, variables of these types are generated by hardware during execution. They are called implicit hardware types: dynamic thread descriptor and message.

Dynamic Thread Decriptor (DTD)

A dynamic thread descriptor contains the information needed for thread execution at run time:

Dsp	Dfp	IBA	Rg	IP	Lg
12	20	20	4	20	8

Dsp is the descriptor frame pointer (base address) to its local descriptor frame, Dfp the data frame pointer to its local data frame, *Rg* the register set number which is allocated when the thread is scheduled for execution, IBA the base address to the code of latency synchronization threads of the current activation, IP the instruction pointer to the next instruction, *Lg* the number of the remaining instructions to be executed. For latency synchronization threads, their dynamic thread descriptors are constructed by the synchronization unit. The details about handling latency synchronization threads will be described in the next section.

Messages

A message is constructed by a split operation and routed to a remote processing element, to the I-structure storage or to a functional unit within the processing element on which the split operation is performed. Messages do not have fixed size, but vary according to the kind of message. An approximation to a message format can be represented as follows:

$$< U, [NODE], [MOP], arg1, arg2, ... >$$

where U says whether this is a message to a remote node or to a functional unit within the same node, $NODE$ is the processing element number or memory module number, MOP specifies how the message should be handled, i.e. operations like $read$ or $write$, the arguments are constructed according to the message operation MOP. [ ] means that the included item may be omitted based on the types of messages. There are generally three types of messages:

1. Messages for local split operations

Whenever a local split operation is encountered, a request message is built and routed to the corresponding functional unit. The format of the request message is:

$$< 0, FU, operand1, operand2, IBA, ls, LST/addr, AD >$$

where the bit *0* indicates that this is a local message, FU the functional unit number, IBA the base address of a memory segment where all latency synchronization threads of the running activation reside, ls indicates whether the response message will initiate a latency synchronization thread or not[1], LST points to the corresponding latency synchronization thread, $addr$ is an offset to the data frame and used for direct writing of the generated result (together with ls) in a data frame location without initiating a latency synchronization thread, and AD references the current descriptor memory and the current data frame.

After execution, the functional unit builds a response message which consists of

$$< 0, IBA, LST, AD, result >$$

which will be routed to the synchronization unit where the corresponding latency synchronization thread is initiated by using IBA und LST. The *result* may then be written into a data frame location or into a global register.

2. Messages for access to remote processing elements

Accessing to remote processing elements mainly performs argument/result transfer between different processing elements. In this case, the destination processing element and the destination activation must be explicitly given by associated instructions. Such messages have the following format:

$$< 1, NODE, IBA, LST, AD, A >$$

where the bit *1* means that this message will be routed to a remote node, the field $NODE$ specifies this remote processing element number, the message operation is *write*[2], A is the data value which is written to the remote processing element, AD an explicit activation descriptor, and IBA, together with LST, identifies the corresponding latency synchronization thread.

3. Messages for access to the I-structure

For access to the I-structure storage, messages are constructed according to the corresponding operations. Messages for read operation generated on the node $NODE1$ have the following format:

$$< 1, NODE0, read, \&A, STP, LST, AD >$$

where $\&A$ represents the address in the memory module $NODE0$. The current processing element number is contained in STP (including NODE1 and IBA). The message returned by the memory module consists of:

[1] ls can be used to group several split operations together, in order to reduce the number of latency synchronization threads to be executed (see the next chapter for more details about that

[2] Note that no *read* message operation is needed for accessing remote processing elements, because no *read* operations are required for argument/result transfer between processing elements. Moreover, the data memory in any processing element does not support synchronizing loads like I-structure memory does.

$$< 1, NODE1, IBA, LST, AD, A >$$

where A is the returned value from the memory module.

Differently, messages for write operation without acknowledgement have the following format:

$$< 1, NODE0, write, \&A, A >$$

However, for write operations returning an acknowledgement to the current executing activation, messages are similar to those for read operations:

$$< 1, NODE0, write, \&A, A, STP, LST, AD >$$

The returned acknowledgement will initiate a latency synchronization by the following message

$$< 1, NODE1, IBA, LST, AD >$$

6.2.3 Storage Organization

The LDME recognizes five storage resources: code memory, data memory, descriptor memory, structure memory and register sets. Except for register sets all other memories are managed by the resource manager at run time. Register sets are allocated or deallocated by hardware whenever an enabled thread is scheduled for execution or an executing thread terminates.

As mentioned, the LDME supports a tree of activations. Hence, many enabled activations may exist on a processor, i.e. many activations may have several currently enabled threads. The fundamental concerns here are *how many* activations should become resident, *where* enabled threads reside and *how much* resource should be allocated for scheduling an enabled thread. Such issues are essential to the design of a multithreaded processor, but have received little attention in the literature. We argue that these are essential, because we can not expect that the entire activation tree of any parallel program could be maintained in a high speed processor storage, even though the activation tree could be reduced with some heuristic algorithms. The research work in **Id, Sisal** etc. has shown that the activation tree of a program may be large [Cul 91a].

In this preliminary design we have chosen **two** resident activations on a processing element. The choice is based on the performance analysis in chapter 4: an eight stage SORLD multithreaded pipeline reaches its peak performance by executing ca. 8 parallel threads under a memory latency of 64 cycles. Observing other previous work [Ian 90], it is reasonable to assume that an executing activation contains on average more than four enabled threads during its execution period, hence two resident activations may contain in average more than 8 threads, which are required to achieve the peak performance of our multithreaded pipeline. Moreover, more resident activations also need more hardware requirements (e.g. more register

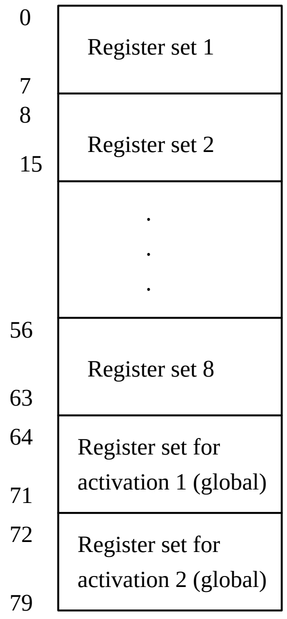


Figure 6.2. The layout of multiple register sets

sets must be provided).

Another essential issue in designing a multithreaded processor concerns multiple register sets. As known, each executing thread references its own register set which is allocated when the thread is scheduled for execution. As usual, the use of one register set for each executing thread can improve the efficiency of code [Pap 91a]. The layout of the register sets is shown in Figure 6.2.

The decision to choose 8 registers for each register set seems to be reasonable based on empirical results from other previous work such as MASA's task frame [Hal 88]. Note that in the LDME a scheduled thread is guaranteed to complete. A thread may only be suspended temporarily whenever a hazard (data or control) is detected. When a thread terminates, its register set is freed for scheduling another enabled thread. Data share accross threads within an activation is performed through the corresponding *global register set*, which, however, must be saved when a resident activation is suspended, and restored when the activation becomes resident again.

6.2.4 Instruction Set Architecture

Instruction Formats

All instructions are presented in a 32 bit word. There are basically two instruction formats, which are used to define the two major kinds of instructions: the instructions with deterministic operation latency and the instructions that initiate split operations. The instruction formats are shown below:

Opcode	OL	mode	reg1	reg2	reg3/literal
6	2	2	4	4	14

Instruction format (a)

Opcode	reg1/ls	reg2	reg3	literal
6	4	4	4	14

Instruction format (b)

The first format is used to define all instructions with deterministic operation latency, which are mostly the normal RISC-instructions. In this format the field **OL** specifies the operation latency[3] of an instruction until its result can be used as an operand for its successive instruction(s) (data manipulation instructions) or its effects allow the execution of the following instruction(s) (branch instructions). The field **mode** defines four different address modes:

1. register direct – the result of the operation on *reg2* and *reg3* is written into *reg1*.
2. immediate – the field *reg3/literal* is a 14 bit literal value, which is used as a relative displacement of branch instructions or a literal operand.
3. data frame direct – *reg2* and *reg3/literal* together form a 18 bit relative address to the actual local data frame.
4. based addressing – *reg1* is used as the base register and *reg3/literal* is a relative address. The effective address is computed by adding the content of the base register and the literal value.

Note that the address modes (3) and (4) are used only for the **load** and **store** instructions. The mode (4) allows frame index access.

The second instruction format is used to represent all instruction initiating split operations. There are further two sub-formats which represent instructions initiating local split operations and remote accesses respectively:

1. local split operations – there are two different cases: the field *ls* is used to define whether the result of a local split operation will initiate a latency synchronization thread or not. In the first case, *reg2* and *reg3* refer to two operands, *literal* refers to the instruction pointer to the corresponding latency synchronization thread (LST). In the second case, *literal* refers to a data frame location. More details about this can be found in the next section.

[3] Note that the operation latency of an instruction here does not refer to the execution time of this instruction, but refers to the time that must be delayed for scheduling its successive instruction to execute.

2. remote accesses – in this case, *reg3* refers to a remote memory module (I-structure access) (NODE) or to the synchronization thread pointer (STP) in a remote processing element, *reg1* refers to the data value for remote write operations, and *reg2* refers to an address for I-structure access or an activation descriptor for an access to a remote processing element. Again, the field *literal* refers to the instruction pointer of the corresponding latency synchronization thread (LST).

Instruction types

In the following, we present a basic instruction set, which may be extended by adding more related instructions. The concerned instructions can be divided into the following groups:

- *Basic RISC instructions* - these include integer arithmetic, logic, predication, shift, load, store, and branch instructions. The address modes of instructions are determined in the field **mode** in the instruction format. The register-register instructions access operands and results only in their own register set. Like other RISC machines, the movement of data between a data frame and a register set is performed through **load** and **store** instructions.

Integer arithmetic instructions	
addi des, op1, op2	integer add
subi des, op1, op2	integer substract
abs des, op1, –	integer abs
Logic instructions	
and des,op1, op2	bitwise and
or des, op1, op2	bitwise or
xor des, op1, op2	bitwise xor
not des, op1, –	bitwise not
Predicate instructions	
eqp des, op1, op2	des = true if op1 = op2
nep des, op1, op2	des = true if op1 $\neq$ op2
gtp des, op1, op2	des = true if op1 $>$ op2
gep des, op1, op2	des = true if op1 $\geq$ op2
ltp des, op1, op2	des = true if op1 $<$ op2
lep des, op1, op2	des = true if op1 $\leq$ op2
Branch instructions	
jump displ.	unconditional jump
tjump pred., displ.	jump if pred. = true
fjump pred., displ.	jump if pred. = false
Load/Store instructions	
load des, [BA], addr	load data from a data frame
store addr, [BA], source	write data to a data frame

(a). Basic RISC instructions

- *Thread instructions* - these are instructions used to initiate or terminate a thread. This group contains the following instructions: **fork** initiates a local thread within the same activation, while **start** initiates a remote thread belonging to a different activation. The instruction **start** is introduced mainly for supporting **loop unrolling** across multiple activations. **tfork** and **ffork** are two conditional fork-thread instructions. The last instruction **stop** terminates the executing thread explicitly. As mentioned in the last chapter, a normal thread is terminated implicitly by checking the field **Lg** in its dynamic thread descriptor. The instruction **stop** allows to terminate threads whose execution length (time) may *not* be statically determined at compile time.

Thread instructions	
fork thr	initiate a local thread
tfork pred., thr	initiate thr if pred. = true
ffork pred., thr	initiate thr if pred. = false
start Dsp, thr	initiate a remote thread
cfork pred., thr1, thr2	initiate thr1 if pred. = true else thr2
stop	terminate the running thread

(b). Thread instructions

- *Code block Call/Return instructions* - these are instructions used to allocate a context by invoking a code block, to transfer arguments and results, and to deallocate the allocated context after execution. Three instruction are included. All three are split-phase instructions and support an asynchronous remote call/return mechanism. **alloc CB, LST** requests a context for the specified code block **CB**, the returned message initiate a latency synchronization thread **LST**, which then saves the pointer of the allocated context into the data frame of the caller. **rwrite A, node:IBA, Dfp,LST** stores the data value A into the remote processing element by initiating a latency synchronization thread **IBA+LST**, which saves A into the corresponding data frame location and initiates further threads. The last instruction **release parent, LST** deallocates the context and signals its parent activation by initiating a latency synchronization thread **LST**.

Code block Call/Return instructions	
alloc CB, LST	allocate an invocation context for CB
rwrite A, node:IBA, DP, LST	write A into a remote data frame
release parent(STP), LST	deallocate the context and signal parent

(c). Call/Return instructions

- *Local split-phase instructions* - we define all floating point instructions, integer multiply and integer divide to be local split-phase instructions. This is because all these operations are performed in separate functional units and the execution time of different operations may vary. The *ls* field of instructions determines whether a local latency synchronization thread will be initiated or not after completion of a split operation. The local latency synchronization thread may save the result into a local data frame location or a global register and then activates other normal threads. The reason for introducing the second operation mode, i.e. the result is directly written into a data frame location, is to increase the length of normal threads and to reduce the number of latency synchronization threads being executed. This mode provides a similar architectural support for explicit-switch as studied in [Boo 92]. The benefit of this mode will be demonstrated in the next chapter. Nine instructions have be defined as follows:

Local split-phase instructions	
multi op1, op2, LST/addr	integer multiplication
divi op1, op2, LST/addr	integer division
fadd op1, op2, LST/addr	floating point add
fsub op1, op2, LST/addr	floating point substract
fmul op1, op2, LST/addr	floating point multiplication
fdiv op1, op2, LST/addr	floating point division
fabs op1, op2, LST/addr	floating point abs
ftoi op1, op2, LST/addr	convert floating point to integer
itof op1, op2. LST/addr	convert integer to floating point

(d). Local split-phase instructions

Meanwhile, any instruction which contains a dynamic execution time or must be performed on a functional unit can be included in this instruction group.

- *Data structure instructions* - these instructions are used to access the I-structure memory. As usual, the data structure operations are split-phase access to the memory module. Every data structure operation causes a request to be generated, which then is routed to the corresponding memory module. The read operation must specify a latency synchronization thread, which is initiated by the corresponding incoming response and saves the response into a specific data frame slot, etc.

  Nine data structure instructions are introduced, of which the first six are the same as defined on TAM [Cul 91a]. Three read instructions **Iread, Ifetch** and **Itake** operate in tandem with three write instructions **Iwrite, Istore** and **Iput**. The instruction **Iread** simply reads the location. Correspondingly, the instruction **Iwrite** simply writes the location. **Ifetch** is a synchronization form of **Iread** and checks status bits (presence bits) associated with the referenced location and waits in the memory unit if the location is empty. **Istore** causes all waiting readers referenced to the location to be serviced. Thus, **Istore** supports write-once data structure with synchronization at element level. **Itake** reads a full location and leaves it empty. If the location is empty **Itake** waits until it is made full by a corresponding **Iput**. Therefore, **Itake** and **Iput** provide exclusive access to a mutable location and are useful to encode critical sections which are essential for implementing resource management functions. The instruction **Iwack** is the same as **Istore** except that an acknowledgement signal for completion is returned and initiates a corresponding latency synchronization thread LST. The acknowledgement can be used, for example, to ensure serial consistency, i.e. the destination thread executes under a guarantee that the store has completed. The last two instructions **Ialloc** and **Irelease** allocate or deallocate a segment in the I-structure memory.

Data structure instructions	
Iread addr, node, LST	read the location
Ifetch addr, node, LST	synchronizing read the location
Itake addr, node, LST	read the location and reset
Iwrite A, addr, node	write the location
Istore A, addr, node	write the location and resume all waiters
Iput A, addr, node	write the location and resume one waiter
Iwack A, addr, node, LST	Istore + acknowledge
Ialloc size, LST	allocate a segment in I-structure memory
Irelease size, addr, node, LST	deallocate a segment in I-structure memory

(e). Data structure instructions

- *Miscellaneous instructions* - this group includes instructions used to access queue structure, to transfer data between different functional units, to construct different data types etc.. Until now, the following instructions have been included:

Queue access instructions	
qread des, QD	read a data element from a queue
qwrite A, QD	write A into a queue
sqh1 des, QD, IP	set queue handler1 (underflow)
sqh2 des, QD, IP	set queue handler2 (overflow)
System instructions	
st des, op1, literal	set type for op1
ext des, op1	extract type field
mkad des, Dsp, Dfp	make activation descriptor
mksp des, node, IBA	make synchronization thread pointer
mktd des, IP, Lg:Sc	make thread descriptor
Direct move instructions	
Dstore A, addr, node	store A into a remote data frame directly
DIread des, addr, node	read I-structure directly
Software trap	
strap vec	software trap with exception vector vec

(f). Miscellaneous Instructions

Two queue instructions are used to access a queue explicitly. The further two instructions set exceptional handlers for a queue. The *event driven control* mechanism is supported, i.e. one of the queue handlers is initiated automatically if an exceptional event is detected in accessing the queue. Two instructions **st** and **ext** are provided to operate on the data type field in the data format. **mkad** and **mksp** are used in invoking a code block - an activation descriptor and a synchronization thread pointer are returned by the *resource manager* to the caller - so that the caller can send arguments to the new invoked activation. The instruction **Dstore** stores a data value directly into a remote data frame without initiating any latency synchronization thread. This instruction is used to improve the efficiency of argument transfer in such cases where *non-strict* execution is not needed. Similarly, **DIread** simply reads a location in the structure memory directly without initiating a latency synchronization thread. The software trap instruction **strap** starts an exception code in the entry by the exception vector **vec**. Software trap instructions are essential for fast linkage to some system kernel routines.

Instructions for the synchronization unit

These instructions are used to program all latency synchronization threads, which are executed on a separate synchronization unit. The synchronization unit can be viewed as a conventional RISC processor, augmented with the following instructions:

Instructions for synchronization unit	
rstore A, Dfp, offset	store A into the location Dfp:offset
start Dsp, thr	initiate a thread in Dsp:thr
stop	terminate the running thread and start the next

(g). Instructions for the synchronization unit

The first instruction saves a data value into a specified data frame or a global register. The second instruction initiates a thread directly regardless of whether the corresponding activation is resident or not. If the corresponding activation is not resident, then the thread descriptor of the enabled thread is directly stored into the thread descriptor queue of this activation. Otherwise, the thread descriptor is stored either in the hardware execution queue or in the thread queue of the resident activation. The last instruction terminates the running thread and initiates the next thread from the message queue of the synchronization unit.

6.2.5 Exceptions

For various reasons, an instruction of a running thread may not be able to complete its operation on the input data. Generally, exceptions in the LDME processor can be generated by the following event:

- **Arithmetic Exception**. An arithmetic operation may result in an overflow or underflow, for example, detecting a zero as divider in a division operation.

- **Illegal Presence State Transition**. This type of exception may arise on accesses to the I-structure, for example, detecting an attempted double-write of an I-structure location.

- **Type Trap**. All values have 4 bit hardware type-tags, which are checked at run time. The type-check mechanism defines the range of types allowed for a given operation.

- **Queue Exception**. A queue access could result in one of two possible exception events: queue underflow or queue overflow.

- **Software Trap**. Software Trap provides a fast way to perform dynamic linkage to the system kernel code.

There are many different approaches to handle exceptions in pipelined machines. Generally, such approaches can be divided into **precise** or **imprecise** methods. The precise model can completely reconstruct the state of the computation (the processor) before the exception occured. In many cases, precise reconstruction of the state of the processor is not possible and not necessary. The imprecise model is easier to implement and requires less hardware support. The challenge in designing both an efficient and correct exception mechanism lies in providing the abilities to easily reconstruct the state of the computation before the exception occured and the rapid dispatching to the appropriate exception handler.

The detailed design of the exception mechanism is not in the scope of this work. Here we claim that the approaches used in modern RISC processors are also applicable for our processor (multithreaded pipeline). Further, there is no mechanism in our multithreaded pipeline to resolve pipeline hazards like in many RISC-processors, hence, less state information of an exception thread need to be saved and later restored in comparison with other RISC processors. However, a special problem occurs in any multithreaded pipeline: at any time there may be up to 8 executing threads in the pipeline, and some or all of them may cause an exception. Therefore, there must be 8 copies of the exception mechanism, indexed by the executing thread number - one for each executing thread.

6.3 Multithreaded Processor Microarchitecture

In this section we present the detailed microarchitecture of our multithreaded processor.

6.3.1 Overall Processor Organization

As seen before, the LDME uses the split-phase transaction as a key technique to hide various latencies. The LDME separates two kinds of threads, i.e. latency synchronization threads and non-latency synchronization (normal) threads. Latency synchronization threads are used only to synchronize the responses of split operations before their results can be used by other threads. More important, latency synchronization threads initiate corresponding threads which use the incoming responses. Therefore, latency synchronization threads provide an **active driven** mechanism in the sense of data flow machines. This is also the key difference of our execution model from several other multithreaded execution models in which access to required data values is **passive**, i.e. a thread is initiated without any knowledge whether its required data has been generated, or whether the thread must be suspended if the referenced data is not available.

Figure 6.3 indicates the overall structure of our multithreaded processor, which consists of two separate, asynchronous processors. The decision to use two separate processing units is based on the following considerations:

- The execution of normal threads should not be interrupted. If latency synchronization threads are executed along with other threads on the same processing unit, then an arriving message could interrupt the execution of normal threads, initiates a latency synchronization thread and resumes normal computation. Because there are limited hardware resources for thread execution, some normal threads may have to be unloaded if no hardware resource for a new latency synchronization thread is available. Such unload of threads is expensive and should be avoided.

- Most latency synchronization threads are short, because their main task is to save the arriving responses and initiate other threads. As shown in chapter 4, the run length of threads can influence the performance of a multithreaded processor. Thus, frequent short threads are likely to be severely disruptive.

- More severely, a latency synchronization may initiate another normal thread which belongs to a non-resident activation. Except for loading this not-resident activation into the processor it is not possible to insert the enabled thread into its thread queue if the latency synchronization thread is executed in the same processor. It is more expensive to unload a resident activation with several executing threads.

A separate, asynchronous processor for responding to arriving messages and processing latency synchronization threads is an efficient solution to the problem above.

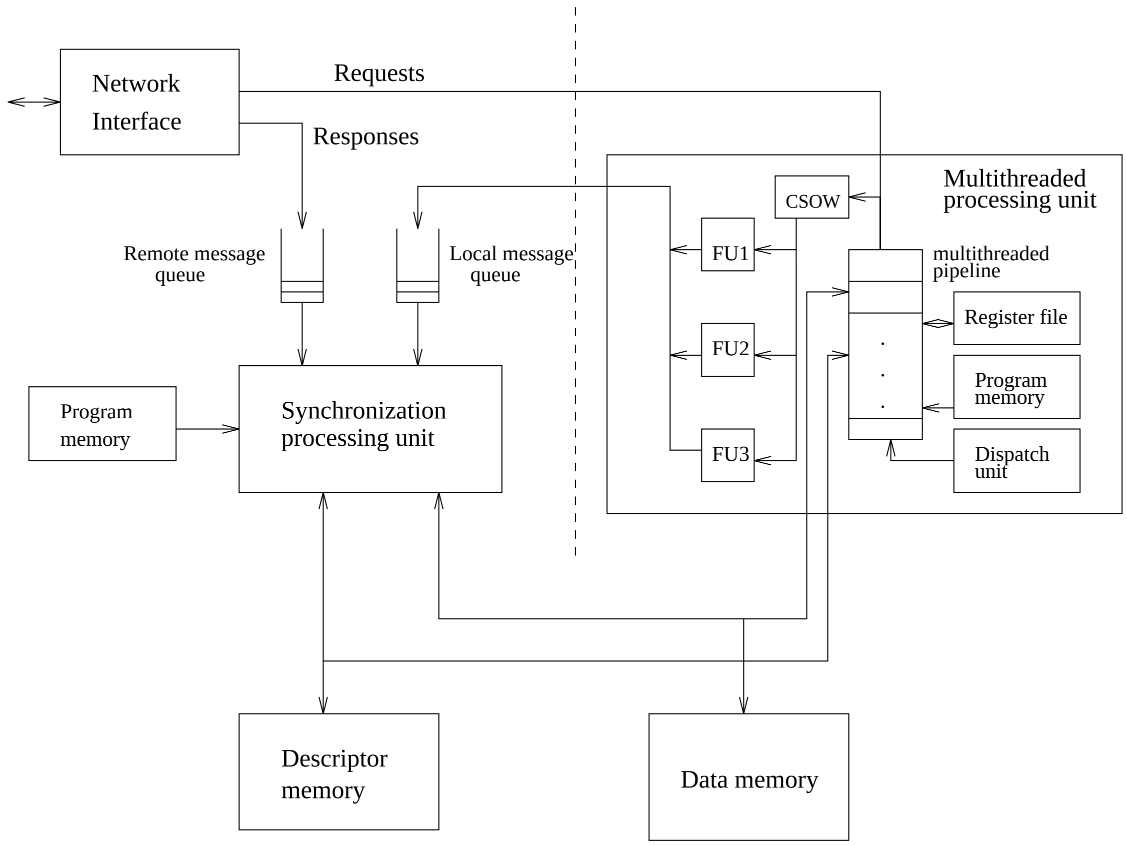


Figure 6.3. The overall structure of the multithreaded processor

6.3.2 The Synchronization Processing Unit

The synchronization processing unit (SPU) is a conventional RISC processor, augmented with some hardware mechanism supporting executing latency synchronization threads. Like RISC processors, the synchronization unit has a program counter, general purpose registers, and a local program memory where latency synchronization threads are stored. The synchronization unit can also access the local descriptor memory and data memory, which are shared with the multithreaded processing unit. Unlike a conventional RISC processor, the synchronization unit must react to incoming messages very rapidly, this includes rapid loading of message data values into registers and rapid starting an associated latency synchronization thread. In the following, we deal with these issues in more details.

Remote and Local Message Queue

The synchronization processing unit has two hardware message queues that have an ordered priority – messages from the lower priority queue (local message queue) are only processed when the higher priority queue (remote message queue) is empty. It should be noted that all incoming messages from the network are always enqueued into the remote message queue in FIFO mode, which is required to preserve the essential FIFO property of the network [Pap 91a]. This remote message queue is similar to the *system queue* on Monsoon.

As described in the last section, there are basically three kinds of messages received by the synchronization processing unit: messages from the local multithreaded processing unit, messages from remote processing elements, and messages from I-structure memory modules. The first kind of messages is generated by functional units after completing their operations and is routed to the local message queue. Using the local message queue we want to avoid the complex hardware mechanisms used to support *out-of-order* execution and to resolve pipeline hazards (structure and data hazards) just like those in advanced pipelines (see chapter 3). The second kind of messages is generated by any remote processing element in transfering arguments or results and is routed to the remote message queue. The last kind of messages is returned from I-structure memory modules and also is routed to the remote message queue.

Message Handling

As mentioned, the synchronization processing unit must react to incoming messages rapidly. Like a dataflow processor, the synchronization processing unit is triggered by the arrival of a message. The description of different message types in the last section has shown that all incoming messages to the synchronization processing unit have a fixed format as follows:

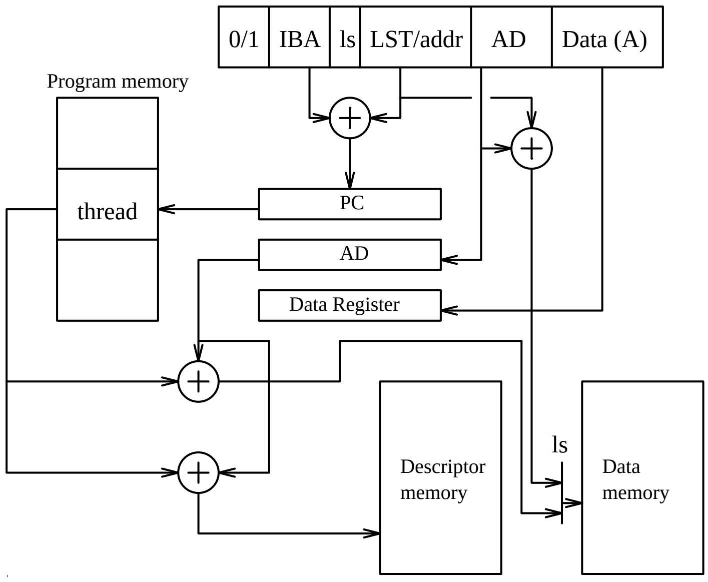


Figure 6.4. The message-handling mechanism in the SPU

0/1	IBA	ls	LST/addr	AD	Data (A)

To handle incoming messages, we propose a simple mechanism as indicated in Figure 6.4. The mechanism can response to an incoming message in one cycle if the message can be read in parallel or three cycles if the message can be read in one word on every cycle.

The synchronization processing unit reserves three special registers: a program counter register, an activation descriptor register, and a data value register. All three registers are directly loaded by the hardware whenever the synchronization processing unit is idle. IBA is the base address of the latency synchronization threads of the activation identified by the corresponding activation descriptor AD. The sum of IBA and the offset LST points to the corresponding latency synchronization thread. The activation descriptor register is used to reference the descriptor memory and data memory, which are shared with the multithreaded processing unit. The incoming data value is loaded in the data register. If no latency synchronization thread will be initiated (indicated by the ls field), then the incoming data value will be directly written into the data frame location $Dfp + addr$.

Latency Synchronization Threads

After reacting to an incoming message, the synchronization processing unit begins with the execution of instructions pointed to by the address in the PC register. The typical latency synchronization thread for a remote read operation is shown below:

```
rstore Rd, Dfp, offset     -- store Rd into Dfp+offset
start Dsp, thr             -- initiate a normal thread
stop                       -- stop and handle the next message
```

The **rstore** instruction saves the data value of the message into the data frame location $Dfp + offset$. The next instruction **start** initiates a thread, which is referenced through $Dsp + thr$ in the descriptor memory. Thus, this thread can use the data value of the message directly. The last instruction **stop** ends the present latency synchronization thread and starts to handle the next message from one of the message queues by reloading its registers from the next message, after which a new latency synchronization thread will be executed.

For a remote write operation with acknowlegement, the latency synchronization thread may be as follows:

```
start Dsp, thr         -- initiate a normal thread
stop                   -- stop and handle the next message
```

The corresponding activation is informed by the thread $Dsp + thr$ that the write operation has completed.

A Comparison to *T Processor

*T is a multithreaded processor, which has been recently proposed by Arvind and his collegues at MIT [Nik 92]. Similar to our LDME processor, *T contains also a asynchronization coprocessor. It is interesting to compare our mechanism to that of *T.

There are several obvious differences between our synchronization processing unit and *T's synchronization coprocessor:

1. In *T there is a hardware queue called *Continuation Queue* between the synchronization coprocessor and the data processor (which corresponds to the multithreaded processing unit in our processor), and the data processor obtains a thread from this continuation queue. In our processor, there is no such continuation queue between the synchronization processing unit and the multithreaded processing unit. Communication between them can only be performed through the shared data memory.

2. In *T any thread from the continuation queue can be executed immediately whenever the data processor is idle. Differently, in the LDME processor, enabled threads reside in a local thread queue. Only threads belonging to a resident activation can be executed.

3. The synchronization coprocessor in *T executes a *join* instruction, which checks a location in the data frame called a *join counter* and causes a thread to be enabled if

the counter reaches the terminal count. In our processor, the synchronization counts for threads are implictly defined in their thread descriptors, which reside in a separate descriptor memory.

From the comparison above, it is obvious that our execution model provides a better locality of activation execution, because only threads belonging to resident activations are scheduled for execution in the multithreaded processing unit. Normal threads belonging to a resident activation are initiated by the multithreaded processing unit and can be immediately executed. Thus our execution model may have a better reaction time for such threads, which is important for preserving locality of activation execution. Further, our model can lead to less access conflicts on the shared data memory, because the synchronization processing unit does not need to access any *join counter* like *T's synchronization coprocessor.

6.3.3 The Multithreaded Processing Unit

We are now in the position to specify the main part of the LDME processor – the multithreaded processing unit (MPU) in more detail. The processor model presented here was extracted from a variety of hardware proposals either for multithreaded processors or for modern RISC processors and fulfills requirements for the LDME model of the previous chapter.

Basic Organization

Figure 6.5 shows a block diagram of the multithreaded processing unit. The unit looks like a modern RISC processor and incorporates two major operations parts, a multithreaded pipeline and functional units. The functional units are used to compute local split operations, which operate asynchronously and communicate data via a hardware window called the **central split operation window**. The pipeline forms a message for each split operation and saves it into the central split operation window. It will be shown later that the central split operation window here is different from either **scoreboard** or **reservation station**, which are more complicated to implement in hardware. A functional unit receives a corresponding split operation from the central window whenever it becomes idle. After execution, a return message is built by using the result and sent to the synchronization processing unit, which then saves the result and initiates a further thread.

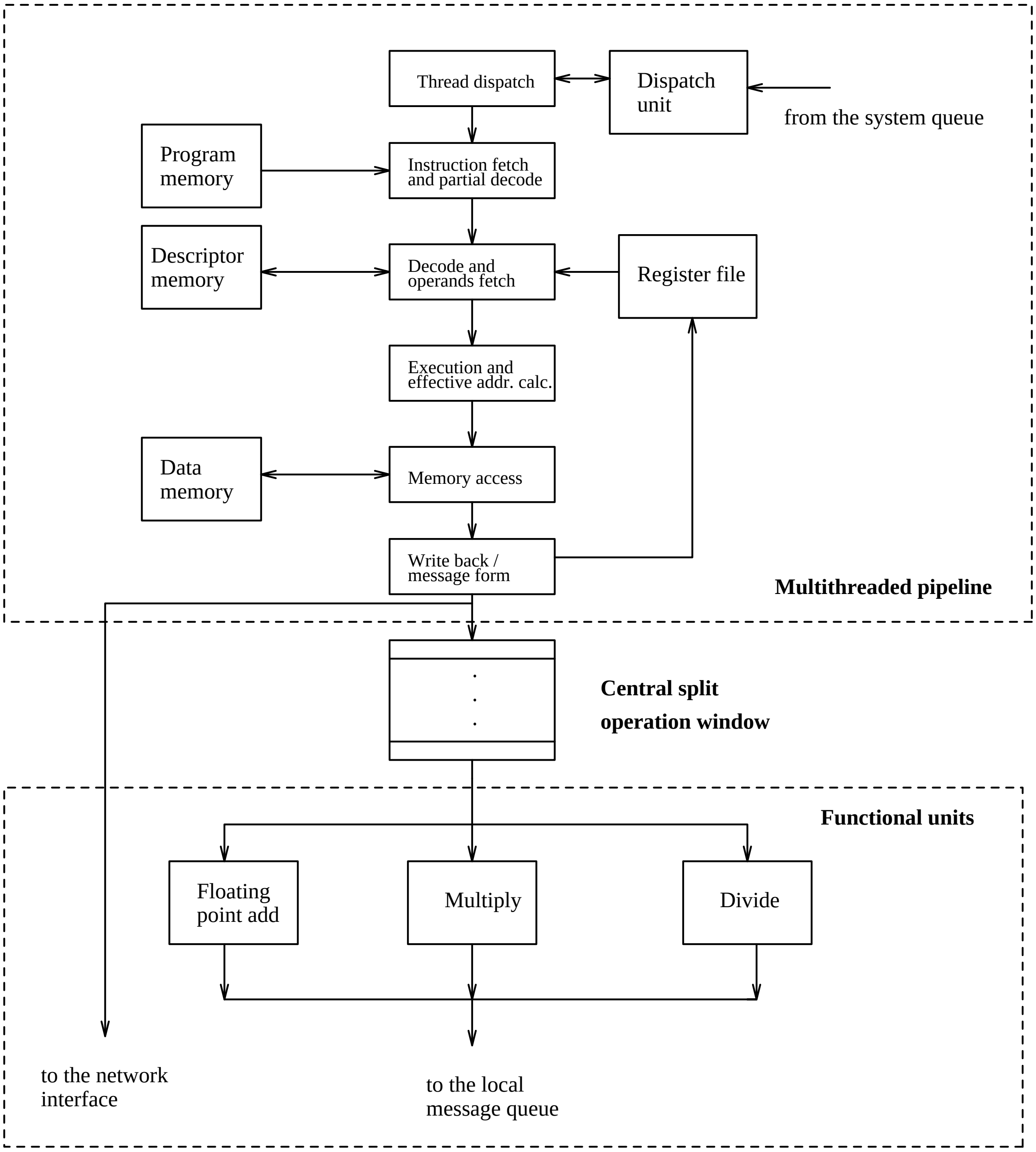


Figure 6.5. The multithreaded processing unit

6.3.4 Functional Units and the Central Split Operation Window

As in many modern microprocessors, three functional units[4] have been included for performing the local split operations defined in the last section. A **Floating point adder** executes all floating point add, substract, absolute, and conversion operations. We integrate integer and floating point multiplys as well as and integer and floating point divide into a single fuctional unit, respectively, namely the **multiply unit** handles all integer and floating point multiplys, while the **divide unit** handles all integer and floating point divides. It is assumed that all functional units take multiple cycles to complete their operations. The execution time of each functional unit is here of little importance.

The central split operation window is used to transfer split operations from the multithreaded pipeline to the functional units. It should be noted that the central split operation window is not a queue in FIFO. The window consists mainly of multiple entries where incoming messages are stored and a simple logic which checks the valid messages and sends them to corresponding functional units. The following steps are taken by the central split operation window during program execution:

- accept messages of local split operations from the multithreaded pipeline and allocate empty window entries for incoming messages. The entries are marked *ready* before messages are issued to be proceeded.
- if a functional unit is free, a message for this functional unit in the window is selected and proceeded by this functional unit. If more than one message is ready for issue, a random message from the ready messages may be selected.
- deallocate the window entry after the located message has been issued, so that this entry may receive a new message in the next cycle.

Figure 6.6 shows the basic structure of the central split operation window and its connections to the functional units. The terms in a window entry have the following meaning: the bit V marks whether the corresponding entry is empty or not, FU is the functional unit number, which is associatively searched, together with the bit V. Other terms are extracted from the local message format. The control logic controls the acceptance of messages into empty window entries and routes a next message to a functional unit if it becomes free. If no empty entry is available for incoming messages, the control logic must notify the multithreaded pipeline by sending a full signal to prevent it from decoding split instructions further. The multithreaded pipeline then switches the running thread and activates another enabled thread with no split

[4] In this design we do not include double precision floating point operations.

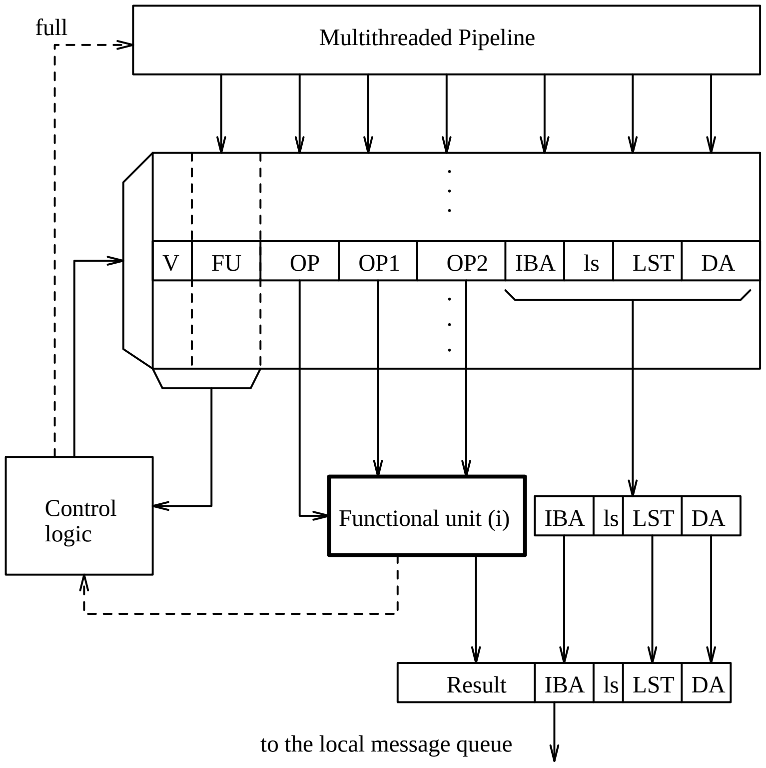


Figure 6.6. The structure of the central split operation window

instructions. The suspended thread becomes enabled when this *full* signal is reset, i.e. when there are empty entries in the window.

One of the novel features of the LDME is that all operations with long or undetermined execution time are split. Therefore, there is no problem with data dependencies in the processor, which are handled in many other pipelined processors through complicated hardware mechanism such as a central reservation station (scoreboard) or distributed reservation stations for detecting and resolving data dependencies. The differences between a reservation station and the central split operation window are obvious:

- the reservation stations are used to handle instructions that have been issued (or decoded) and are waiting for execution, whereas the central split operation window holds messages that are waiting for being issued to the corresponding functional units.
- the reservation stations detect and resolve data dependencies among issued instructions, therefore, the reservation stations include a dependency mechanism. There are no data dependencies between messages in the central split operation window, a message can be issued for execution if the corresponding functional unit is free.
- the reservation stations are proposed mainly for *out-of-order issue* of instructions from the single instruction stream, while the central split operation window is a temporary

storage where messages are waiting for further proceeding (i.e. avoiding the *resource conflict* on functional units).

Thus, the implementation of the central split operation window requires only simple hardware support.

6.4 Multithreaded Pipeline

Now we describe the central part of our multithreaded processing unit - the multithreaded pipeline. First, the basic requirements of the pipeline for supporting the LDME model will be outlined, then the detailed orgranization and the associated operations are described. The multithreaded pipeline presented here exploits the features of the LDME model and avoids complicated mechanisms used in conventional pipeline for detecting and resolving pipeline hazards. The new mechanisms used for the multithreaded execution are easy to implement. The multithreaded pipeline is, therefore, claimed to be realizable in hardware.

6.4.1 Basic Requirements

As described earier, our multithreaded pipeline supports execution of eight threads in parallel to exploit inter-thread parallelism. Moreover, intra-thread parallelism is also exploited to improve single thread performance and to reduce the required hardware contexts. The design principle is thus to extract intra-thread parallelism, but relies only on inter-thread parallelism as a last resort, because this promises the delivery of optimal performance for as few hardware supports as possible. Two basic requirements are hence obvious:

- a means for **prefetching** a collection of instructions from a subset of enabled threads, **identifying** the set of enabled instructions for execution. This includes checking pipeline hazards and making the suspended instructions enabled if their hazard constraints have disappeared.
- a means for **fast** and **efficient** thread context switching. Because the context switching between executable threads (i.e. threads that possess the hardware resource such as register sets) occur frequently, any extra overhead of the context switching will lead to degrade pipeline performance greatly.

At the instruction level, the multithreaded pipeline issues an instruction for execution like conventional pipelines, i.e., an instruction is issued if no pipeline hazard constraints are met. Unlike conventional pipelines, no attempt to resolve the pipeline hazards by hardware (e.g.

interlocks for data dependency and *squashing* for control dependency) is taken, if any hazard for further issue of instructions from the same thread is detected, but the pipeline suspends the instruction and switches to another executable thread. The hard problem is to know which of the instructions prefetched from corresponding enabled threads are actually executable.

Efficient means are available for detecting pipeline-hazards in many pipelined processors. Differently, we attach the dependency information into each instruction (see the instruction format in 6.2.4. The idea of *tagged* instructions is to detect pipeline hazards among instructions from a thread under the following considerations:

- all static dependencies among instructions are easily generated through simple data flow analysis, which belongs to one of the most successful techniques used in RISC-compilers[5].
- Using *tagged* information allows the multithreaded pipeline to detect pipeline hazards arising from the just fetched instruction early, namely, once the instruction is fetched from the program memory. In a conventional pipeline, however, several instruction dispatch cycles may pass between the time when a potentially suspensive instruction is initiated until it can be determined that the instruction will actually suspend. Early detection of any pipeline hazards promises to avoid flushing of not one, but many instructions, depending on the pipeline depth.
- More important, *tagged* instructions greatly simplify the hardware for detecting pipeline hazards. The pipeline hazards are checked immediately after fetching an instruction and a simple decoding allows the dispatcher to know if the instruction will suspend. No further check during execution is required. An instruction is scheduled to execute only if its operands are available. In this sense, the scheduling of instructions for execution is *data-driven*.

6.4.2 The Overview of the Multithreaded Pipeline

Given the above considerations, a multithreaded pipeline has been constructed as shown in Figure 6.7, which is extended from the well known RISC-processor DLX, augmented with the mechanisms for supporting the LDME execution. We use a form similar to one in [Ian 90] to describe the structure of our multithreaded pipeline.

As shown, the pipeline consists of six stages as follows:

1. thread dispatch
2. instruction fetch and partial decode

[5]No dynamic hazards will arise in the LDME model, because all such instructions are split during execution.

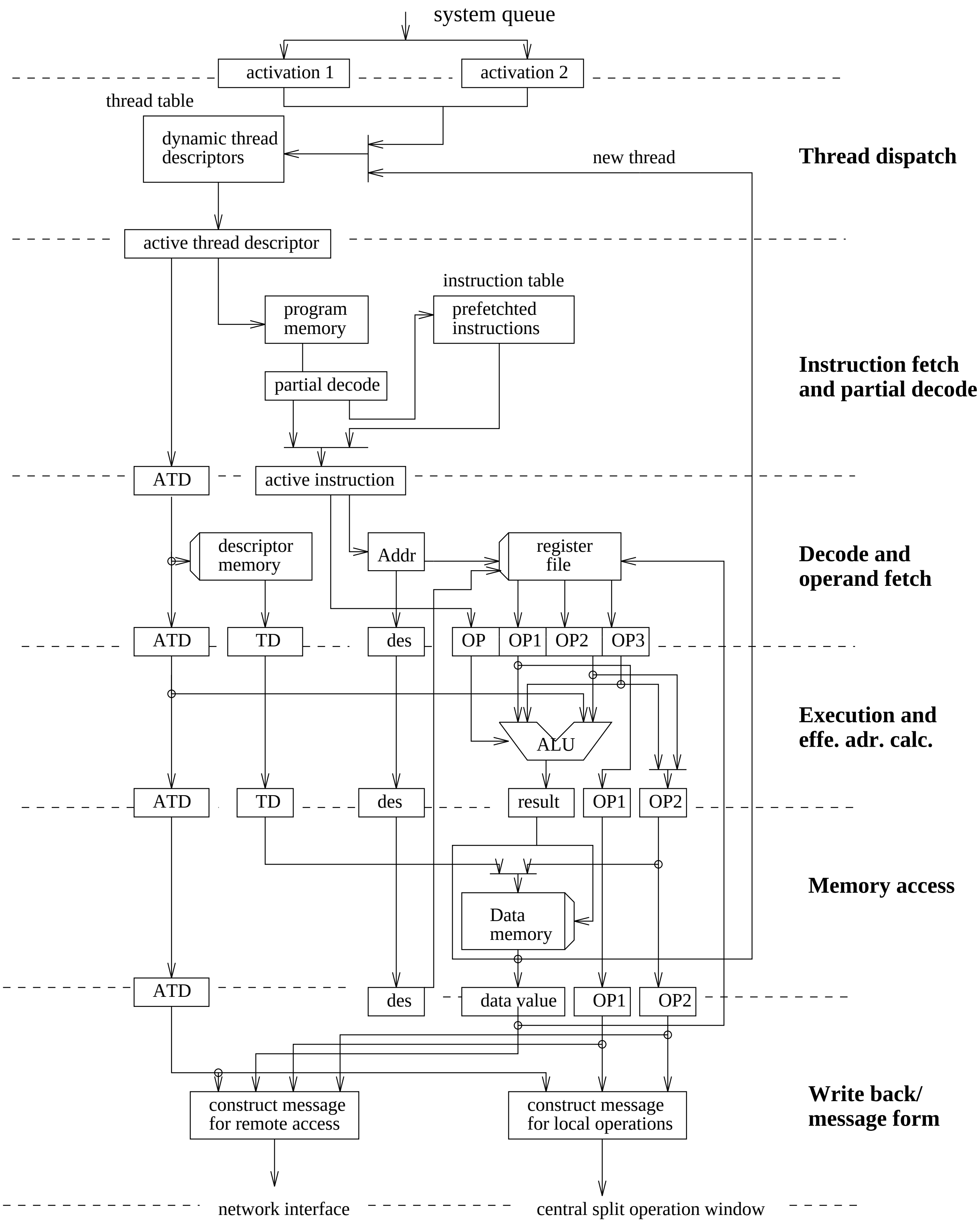


Figure 6.7. The overall structure of a multithreaded pipeline

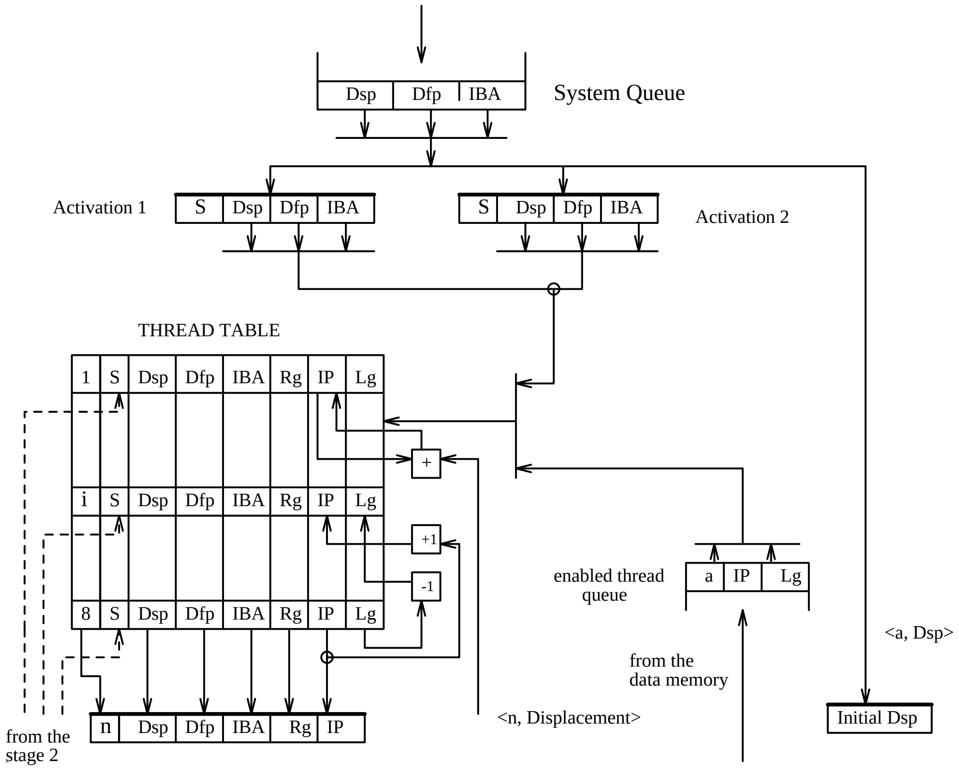


Figure 6.8. Pipeline stage 1 - thread dispatch

3. decode and operand fetch
4. execution and effective address calculation
5. data memory access
6. write back or message form

The pipeline is synchronous, with registers serving as inter-stage interfaces. In the figure, registers are depicted as short rectangular boxes. Stage boundaries are further emphasized with dashed lines. At every pipe beat (clock cycle), each stage stores its current outputs into the registers forwarding to the next stage. In the following, we present the detailed description of the internal data paths in each stage.

6.4.3 Thread Dispatch

The first stage - thread dispatch, as its name implies, is responsible for managing the executable threads and for dispatching one of the threads for execution. Figure 6.8 shows the data paths of this stage in detail. To depict propagation of information in the pipeline clearly, all internal registers are explicitly depicted as short rectangular boxes with a heavy top-bar.

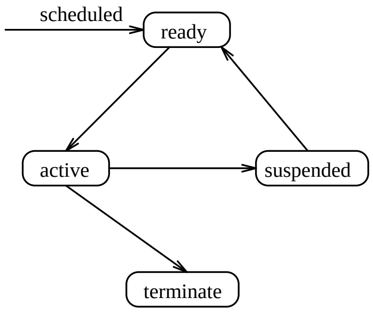


Figure 6.9. Stage diagram of an executable thread

There are two resident activations, which are loaded from the system queue where all enabled activations for this processing element reside. The next activation will be loaded if one of these two activations are terminated or suspended. Once a new activation is scheduled for execution, its descriptor frame pointer Dsp, together with the number of resident activation (1 or 2), is stored in the register $InitialDsp$, which will be used to fetch the thread queue descriptor in the stage 3. We assume that the system queue is managed by the run-time manager.

The thread table contains eight entries, each of which holds a dynamic thread descriptor. This corresponds to the assumption of the previous section that eight executable threads are executed in multithreading. The field S in the activation indicates the state of the thread, which is depicted in the Figure 6.9.

Note the terms used here: we say an enabled thread to be **executable** if the processor resource is allocated for its execution. An executable thread may be in one of four states as shown in Figure 6.9. A thread is said to be **ready** if it has not been scheduled for execution although it contains no constraints (pipeline hazards). At any time, the dispatching logic schedules only one thread to execute, which then becomes **active**. An active thread remains active until a certain hazard arises, which is detected at the stage 2 through the *partial decode*, or the thread has completed. In the former case, the thread is **suspended**, while in the latter case, the thread becomes **terminated** and frees its processor resource.

We represent the dispatching rule in VHDL[6] as follows:

```
ALGORITHM 1

 architecture BEHAVIOR of THREAD_DISPATCH is
   type DYNAMIC_THREAD_DESCRIPTOR is (s, Dsp, Dfp, IBA, Rg, IP, Lg);
   type ACTIVE_THREAD_DESCRIPTOR is (n, Dsp, Dfp, IBA, Rg, IP);
   type THREAD_TABLE is array (8 downto 1) of DYNAMIC_THREAD_DESCRIPTOR;
```

[6]VHDL, short for The VHSIC (Very High Speed IC) Hardware Description Language, is a formal notation for hardware description, standardized by the IEEE in 1987. The purpose of using VHDL here is its ability to describe parallel activities at high level.

```
type ENABLED_THREAD is (a, IP, Lg);
type ENABLED_THREAD_QUEUE is array <> of ENABLED_THREAD;
signal DTD: THREAD_TABLE;
signal ATD: ACTIVE_THREAD_DESCRIPTOR;
signal ETQ: ENABLED_THREAD_QUEUE;
signal no_thread_active : boolean <= true;

begin
       Load_new_thread: process
          begin
            wait until clk'event and clk = '1';
              for i in 1 to 8 loop
                if (DTD(i).s = "empty") then
                   if (ETQ(1).a = 1) then
                     DTD(i) <= (ready, Dsp1, Dfp1, IBA1, Rg, IP, Lg);
                   else
                     DTD(i) <= (ready, Dsp2, Dfp2, IBA2, Rg, IP, Lg);
                   end if;
                   pop_ETQ();
                end if;
               end loop;
           end process Load_new_thread;

        Schedule_next_thread:process
           variable TDT_temp : DYNAMIC_THREAD_DESCRIPTOR;
           variable j: integer := 0;
           begin
              wait until clk'event and clk = '1';
                if (no_thread_active = true ) then
                  for i in 1 to 8 loop
                    TDT_temp := next_thread(DTD);
                    j:= entry_number(TDT_temp);
                    if (TDT_temp.s = "ready") then
                       ATD.n <= j;
                       ADT.Dsp   <= DTD(j).Dsp;
                       ADT.Dfp   <= DTD(j).Dfp;
                       ADT.IBA   <= DTD(j).IBA;
                       ADT.Rg    <= DTD(j).Rg;
                       ADT.IP    <= DTD(j).IP;
                       DTD(j).IP <= DTD(j).IP + 1;
                       if (DTD(j).Lg = 0) then
                          DTD(j).s <= "empty";
                          no_thread_active <= true;
                       else
                          DTD(j).Lg <= DID(j).Lg - 1;
                          no_thread_active <= false;
                       end if;
                       exit;
                     end if;
                   end loop;
                 else
                    k = active_thread_number(DTD);
                    if (DTD(k).Lg = 0) then
                       DTD(k).s <= "empty";
                       no_thread_active <= true;
                    else
                       DTD(k).IP <= DTD(k).IP + 1;
```

```
                    DTD(k).Lg <= DTD(k).Lg - 1;
                    no_thread_active <= false;
                  end if;
                end if;
            end process Schedule_next_thread;

        Branch:process
            begin
              wait until clk'event and clk = '1';
                if (branch = true) then
                  DTD(n).IP <= DTD.(n).IP + Displacement
                end if;
            end process Branch;

        State_modify:process
            begin
              wait until clk'event and clk = '1';
                for i in 1 to 8 loop
                  if (INST(i).s /= "empty") then
                    if (INST(i).s = "hazard") then
                      no_thread_active <= true;
                      DTD(i).s <= "suspended";
                    else
                      if (DTD(i).s = "suspended") then
                        DTD(i).s <= "ready";
                      end if;
                    end if;
                  end if;
                end loop;
            end process State_modify;

  end BEHAVIOR;
```

The VHDL description consists of four processes, which are executed in parallel. The first process *Load_new_thread* attempts to find an empty entry in the thread table. If an empty entry exists, the next enabled thread from the enabled thread queue will be stored into this entry, which is then masked as **ready**. The second process attempts to schedule a ready thread to execute if the current active thread has to be suspended, otherwise the active thread remains active. *next_thread* and *entry_number* are two procedures used to select the next thread descriptor and to determine the corresponding entry number in the thread table. The third process computes the jump destination of branch instructions. The last process modifies the states in the thread table. The control signal for modifying states (INST(i).s) stem from the stage 2, because the pipeline hazards can only be determined after an instruction has been fetched.

By this method, an active thread remains scheduled for execution until certain pipeline hazard is encountered or the thread is terminated. This reflects our design principle that first the intra-parallelism should be extracted, and the inter-thread parallelism, as a last resort, is used to hide various latencies. No extra overhead for context switching is required. If a hazard is detected in the active thread, the next ready thread will be scheduled for execution in the

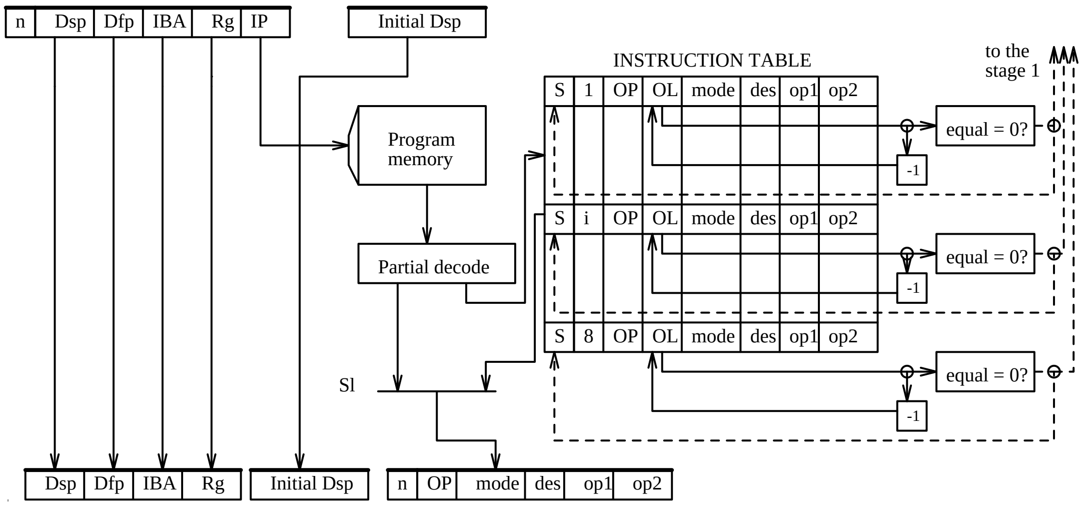


Figure 6.10. Pipeline stage 2 - Instruction fetch and partial decode

following clock cycle.

6.4.4 Instruction Fetch and Partial Decode

The second pipeline stage is reponsible for fetching an instruction from the active thread, checking for pipeline hazards by a simple partial decode. Its detailed structure is shown in Figure 6.10.

The field OL in the fetched instruction contains hazard information: no hazard if $OL = 0$. Otherwise, the fetched instruction must be switched into the instruction table and wait for rescheduling. A fetched instruction may be in one of three states. An instruction is **ready** if its hazard constraint has disappeared, while an instruction with $OL > 0$ is said to be **suspended**. An instruction becomes **active** if it is scheduled for execution. If an active instruction is extracted from the instruction table (see Figure 6.10), the corresponding entry must be masked as **empty**.

Again, we use a VHDL decription to define the instruction scheduling algorithm as follows:

```
ALGORITHM 2

 architecture BEHAVIOR of INSTRUCTION_DISPATCH is
   type INSTRUCTION is (s, n, OP, OL, mode, des, op1, op2);
   type INSTRUCTION_TABLE is array (8 downto 1) of INSTRUCTION;
   signal ATD: ACTIVE_THREAD_DESCRIPTOR;
   signal INST: INSTRUCTION_TABLE;
   signal INST_REG, AI: INSTRUCTION;
   signal no_instruction_active : boolean <= true;
```

```
begin
       Read_instruction: process
          begin
            wait until clk'event and clk = '1';
              if (no_thread_active = false) then
                INST_REG <= (n, PROGRAM_MEMORY(IP));
              else
                INST_REG.s <= "empty";
              end if;
          end process Read_instruction;


        Partial_decode:process(INST_REG)
           begin
                 if ( INST_REG.s \= "empty" ) then
                    if (INST_REG.OL \= 0 ) then
                       INST(n) <= INST_REG;
                       INST(n).s <= "suspended";
                       no_instruction_active <= true;
                    else
                       AI <= INST_REG;
                       no_instruction_active <= false;
                    end if;
                 else
                    no_instruction_active <= true;
                 end if;
           end process Partial_decode;

        State_modify: process
           variable INST_temp : INSTRUCTION;
           variable j: integer := 0;
           begin
               wait until clk'event and clk = '1';
                 for i in 1 to 8 loop
                     INST_temp := next_instruction(INST);
                     j := entry_number(INST);
                     if (INST_temp.s = "ready") then
                        if (no_instruction_active = true) then
                           AI <= INST(j);
                           INST(j) <= "empty";
                        end if;
                     else
                        if (INST_temp.s = "suspended") then
                           if (INST_temp.OL > 0) then
                              INST(j).OL <= INST(j).OL - 1;
                           else
                              INST(j).s <= "ready";
                           end if;
                        end if;
                     end if;
                 end loop;
           end process State_modify;
end BEHAVIOR;
```

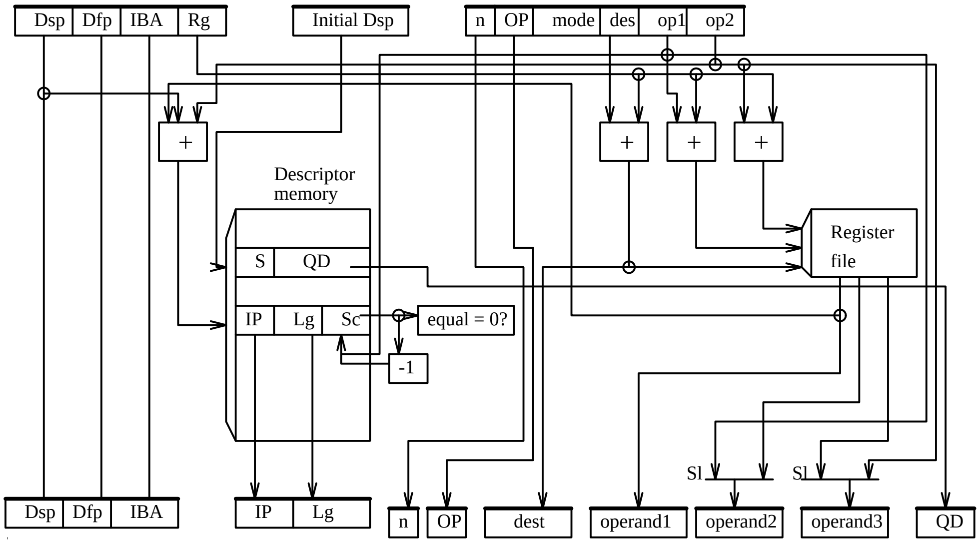


Figure 6.11. Pipeline stage 3 - Decode and operand fetch

The above VHDL description consists of three parallel processes. In the first process, an instruction is fetched from the program at every clock if there is an active thread descriptor forwarded from stage 1. The second process determines whether the fetched instruction will result in a pipeline hazard by checking the field OL. If $OL > 0$, the fetched instruction will be saved into the corresponding entry of the instruction table, and a ready instruction will be extracted instead as the active instruction. Note that this is a switch to an instruction belonging to another ready thread. The last process modifies the states for each instruction in the instruction table, and sends the control signals to stage 1, which are then used to manipulate the state status of the corresponding threads.

By this approach, instructions from the active thread are attempted to schedule for execution unless a hazard is detected by the partial decode. The thread switch and instruction scheduling occur synchronously, no extra overhead is required[7].

6.4.5 Decode and Operand Fetch

In this stage, the active instruction is further decoded, and the corresponding operands are fetched either from the thread's register set or from its descriptor frame. The detailed structure of this stage is shown in Figure 6.11.

As mentioned in the last chapter, a descriptor frame consists of a thread queue descriptor

[7] To demonstrate the instruction scheduling rule clearly, we have ignored the situation when the central split operation window between the multithreaded processing unit and synchronization processing unit becomes full. It is easy to define this situation. The basic approach of avoiding the stall of local split operations is to stop the scheduling of further split instructions, i.e. to make them suspended. Such instructions will be resumed to ready if the central window has empty entries.

pointing to the thread queue and all static thread descriptors of the corresponding activation. The state bits S, attached to a thread queue descriptor QD, indicate whether the corresponding activation is resident (active) or not (ready). When an enabled activation is scheduled for execution from the system queue, the thread queue descriptor of a resident activation is fetched from the descriptor memory by using the *Initial Dsp* register and then stored into a special thread descriptor register in stage 4. The bits S are used to control the access to the thread queue through the synchronization processing unit: if S is active, the synchronization processing unit computes the address of the thread queue by using the thread queue descriptor register in the next stage, otherwise the queue descriptor in the descriptor frame is fetched, modified, and written back. We use the following VHDL description to define operations on the thread descriptor memory:

```
ALGORITHM 3

 architecture BEHAVIOR of DESCRIPTOR_MEMORY is
   type QUEUE_DESCRIPTOR is (s, head, tail, length);
   type THREAD_DESCRIPTOR is (IP, Lg, Sc);
   type DESCRIPTOR_MEMORY is array <> of (THREAD_DESCRIPTOR, QUEUE_DESCRIPTOR);
   type ACTIVE_THREAD_DESCRIPTOR is (Dsp, Dfp, IBA);
   type NEW_THREAD is (IP, Lg);
   signal DSM : DESCRIPTOR_MEMORY;
   signal QD : QUEUE_DISCRIPTOR;
   signal NT: NEW_THREAD;
   signal AT: ACTIVE_THREAD_DESCRIPTOR;

   begin
          Initiate_activation: process
             begin
               wait until clk'event and clk = '1';
                 if (InitialDsp.s = "initial") then
                    QD <= DSM(IDSP.Dsp);
                    QD.s <= "initial";
                    DSM(InitialDsp.Dsp).s <= "active";
                  end if;
             end process Initiate_activation;

          Initiate_thread:process
              variable TD_temp : THREAD_DESCRIPTOR;
              variable DSM_addr: integer;
              begin
                 wait until clk'event and clk = '1';
                  case (OP) is
                    when "fork" | "cfork" | "start" =>
                       if (OP = "fork" or OP = "cfork") then
                         DSM_addr := Dsp + op2;
                       else
                         DSM_addr := operand1 + op2;
                       end if;
                       TD_temp := DSM(DSM_addr);
                       if (TD_temp.Sc /= 0 ) then
                          DSM(DSM_addr).Sc <= TD_temp.Sc - 1;
                          NT.s <= "no_active";
```

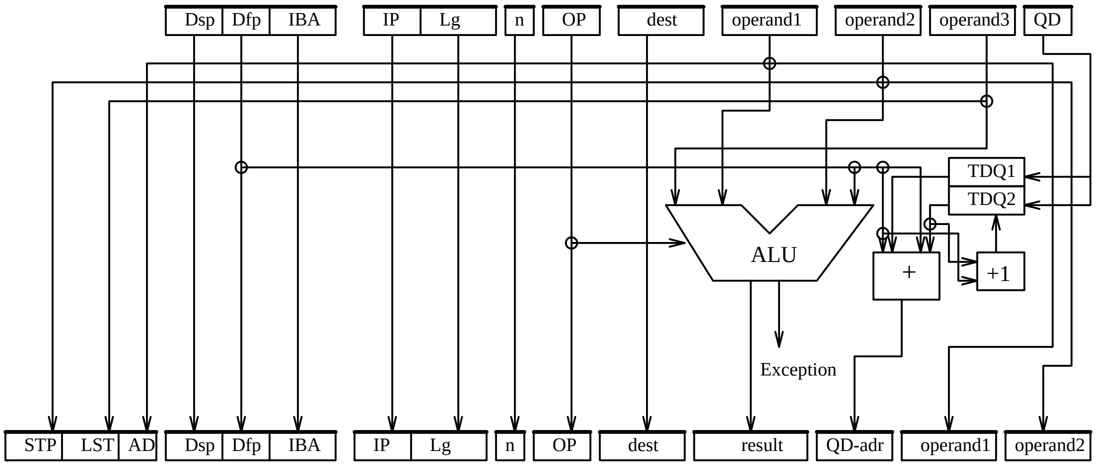


Figure 6.12. Pipeline stage 4 - Execution and effective address calculation

```
                    else
                       DSM(DSM_addr).Sc <= op1;
                       NT.s  <= "active";
                       NT.IP <= TD_temp.IP;
                       NT.Lg <= TD_temp.Lg;
                    end if;
                 when others => null;
               end case;
           end process Initiate_thread;
end BEHAVIOR;
```

The first process is executed only when an activation is scheduled from the system queue. The second process is responsible for initiating a new thread by the instructions *fork, cfork* and *start* (see the instruction set in the last section). The field Sc is called **synchronization count** and is decremented on execution of each initiating instruction on it, and is reset to the original value when the thread is initiated.

6.4.6 Execution and Effective Address Calculation

The actual operations of the active instruction or the calculation of the effective address used to access the data memory are performed in this stage of the pipeline. Figure 6.11 shows its internal structure.

Two register TDQ1 and TDQ2 are used to hold two thread queue descriptors of the corresponding resident activations. The calculation of the address to the thread queue is performed by a separate adder in order to support implicit reading of enabled threads forwarded to the first stage, i.e. prefetch enabled threads.

The actual control logic of this stage is defined in the following VHDL description:

```
ALGORITHM 4

 architecture BEHAVIOR of EX_AND_ADDR is
   type QUEUE_DESCRIPTOR is (s, a, head, tail, length);
   type WORD is bit_vector(31 downto 0);
   signal TDQ(2 downto 1) : QUEUE_DISCRIPTOR;
   signal RESULT: WORD;

   begin
          Access_thread_queue: process
             begin
               wait until clk'event and clk = '1';
                 if (QD.s = "initial") then
                    TDQ(QD.a)   <= QD;
                    TDQ(QD.a).s <= "active";
                 else
                    if (OP = "fork" or OP = "cfork" or OP = "start") then
                        QA_addr <= TDQ(QD.a).tail + Dfp;
                        TDQ(QD.a).tail <= TDQ(QD.a).tail + 1 ;
                                                 -- write thread queue
                    else
                        if (empty(ETQ) = false and
                                empty(TDQ(QD.a)) = false) then
                        QA_addr <= TDQ(QD.a).head + Dfp
                        TDQ(QD.a).head <= TDQ(QD.a).head + 1 ;
                                                 -- implicit read
                        end if;
                    end if;
                  end if;
            end process Access_thread_queue;

            Execution: process
               begin
                 case (OP) is
                   when "memory access" =>
                      RESULT <= Dfp + operand3;   -- the address
                   when ("arithmetic" | "logic" | "predicate") =>
                      RESULT <= operand1 op operand2;
                   when others => null;
                  end case;
               end process Execution;
   end BEHAVIOR;
```

As seen, the first process is responsible for the access to the thread descriptor queue in the data frame. Note that enabled threads are fetched implicitly by the hardware to fill the enabled thread queue in the first stage. Implicit reading of thread descriptors is exclusive to writing of thread descriptors in the thread queue. *empty* is a procedure that determines whether a queue empty or not. The second process performs the normal RISC instructions.

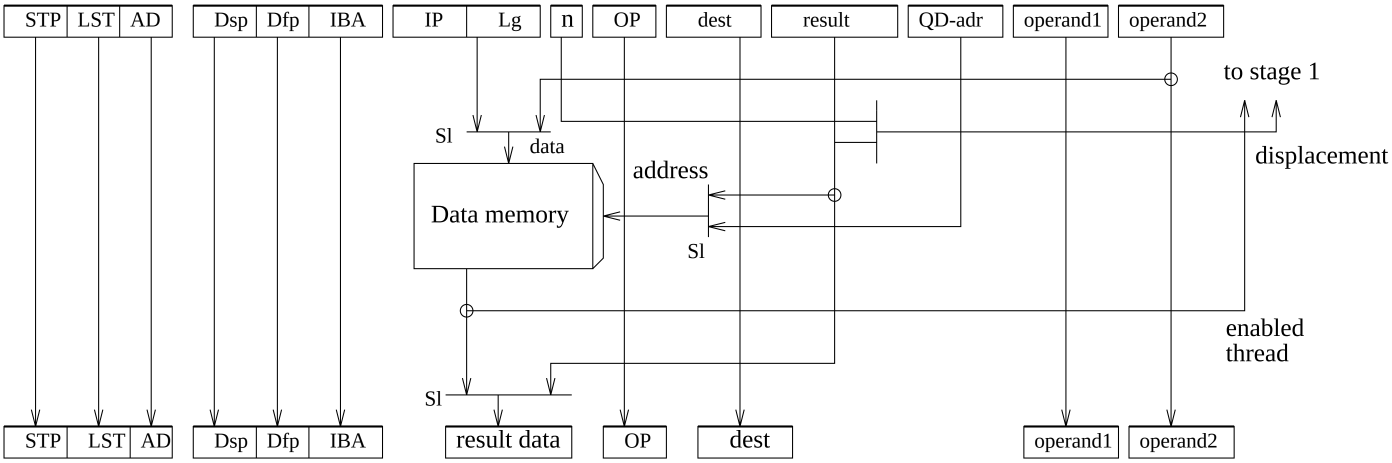


Figure 6.13. Pipeline stage 5 - Memory access

6.4.7 Memory Access

In this stage all accesses to the local data memory are performed (Figure 6.13). Further, the jump destination of a branch instruction is also constructed here, which is then forwarded directly to the first pipeline stage. All implicitly fetched thread descriptors are routed directly to the enabled thread queue in the first stage.

The control logic of this stage can be simply defined in the following VHDL decription:

```
ALGORITHM 5

 architecture BEHAVIOR of MEMORY_ACCESS is
   type WORD is bit_vector(31 downto 0);
   type DISPLACEMENT is (n, displ);
   type DATA_MEMORY is array <> of WORD;
   signal DISP : DISPLACEMENT;
   signal DM : DATA_MEMORY;
   signal RESULT_DATA: WORD;

   begin
          Access_data_memory: process
             variable DM_addr : integer;
             variable DMR : WORD;
             begin
               wait until clk'event and clk = '1';
                 case (OP) is
                   when "load" =>
                       DM_addr := result;
                       RESULT_DATA <= DM(DM_addr);
                   when "store" =>
                       DM_addr := result;
                       DM(DM_addr) <= operand2;
                   when "fork" | "cfork" | "start" =>
                       if (NT.s = "active") then   -- a new thread
                          DM_addr := QD_adr;
```

```
                    DM(MD_adr) <= NT         -- store a TD
                 end if;
             when others =>
                 if (empty(TDQ(QD.a)) = false ) then
                   DM_addr := QD_adr;
                   ETQ <= DM(DM_adr);       -- implict read a TD
                 end if;
           end case;
     end process Access_data_memory;

     Branch_instruction:process
        begin
           wait until clk'event and clk = '1';
            if  (OP = "branch instructions" ) then
                 DISP <= (n, result);
                 send_DISP_to_stage1();
            end if;
        end process Branch_instruction;

end BEHAVIOR;
```

Note that implicit reading of the thread descriptor queue is performed only if no other accesses to the data memory occur. The implicitly fetched thread descriptors are then routed directly to the first stage. The second process determines the jump displacement of a branch instruction; the thread number n in DISP is used to identify the corresponding entry in the thread table (see the first stage).

6.4.8 Write Back/Message Form

At the last phase of instruction execution, the result is stored in the corresponding register set, or an associated message is built if a split instruction is encountered (see Figure 6.14).

The control logic of this stage is decribed as follows:

```
ALGORITHM 6

 architecture BEHAVIOR of WRITE_OR_MESSAGE is
   type LOCAL_MESSAGE is (0, FU, OP, operand1, operand2, IBA, LST, Dfp);
   type REMOTE_MESSAGE is (1, OP, STP. LST, Dsp, Dfp, A, &A);
   type WORD is bit_vector(31 downto 0);
   type REGISTER_FILE is array (79 downto 0) of WORD;
   signal LMES : LOCAL_MESSAGE;
   signal RMES : REMOTE_MESSAGE;
   signal REG_FILE : REGISTER_FILE;

   begin
          Write_or_Build_message: process
             variable DM_addr : integer;
             variable DMR : WORD;
```

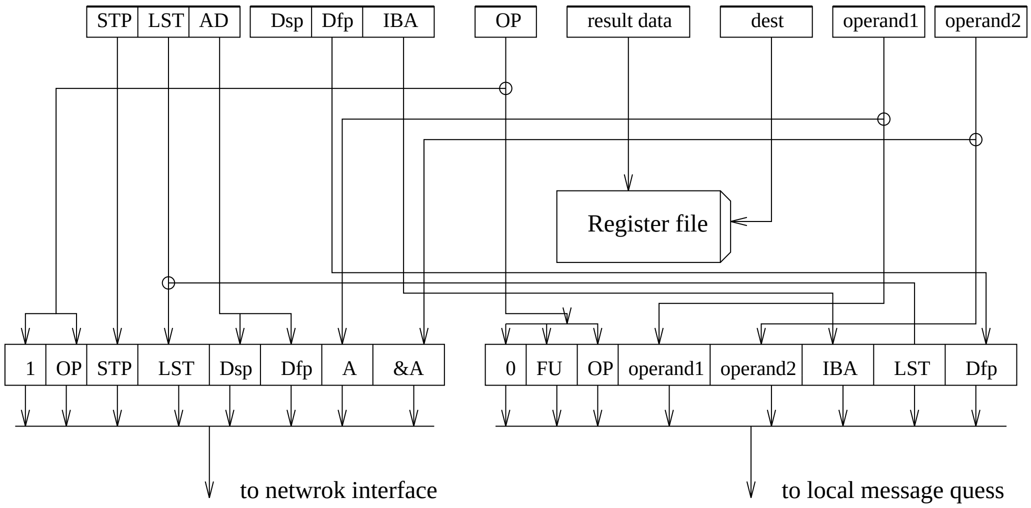


Figure 6.14. Pipeline stage 6 - Write back/message form

```
        begin
          wait until clk'event and clk = '1';
            case (OP) is
              when "write back" =>
                  REG_FILE(dest) <= result_data;
              when "local message" =>
                  LMES <= (0, OP_fu, OP_op, operand1, operand2,
                           LST_iba, LST, Dfp);
                  send_LMES_TO_LOCAL_MESSAGE_QUEUE();
              when "I-structure_read" =>
                  RMES <= (1, read, STP, LST, AD_dsp, AD_dfp,
                           operand1, operand2);
                  send_RMES_TO_NETWORK_INTERFACE();
              when "I-structure_write" =>
                  RMES <= (1, write, STP, 0, 0, 0,
                           operand1, operand2);
                  send_RMES_TO_NETWORK_INTERFACE();
              when "remote_PE" =>
                  RMES <= (1, write, STP, LST, AD_dsp, AD_dfp,
                           operand1, operand2);
                  send_RMES_TO_NETWORK_INTERFACE();
              when others => null
            end case;
        end process Write_or_build_message;
end BEHAVIOR;
```

Note that the NODE number of a remote processing element or a memory module is contained in the STP (see the definitions of data types in the last section). Note also the difference between a message for an I-structure read and a message for an access to a remote processing element. The former contains the local IBA and LST, whereas the latter contains the remote IBA and LST. All local messages are routed to the central split operation window, while all remote messages are routed to the network interface.

6.5 Summary and Remark

The complete design of a LDME processor has been presented, which supports the LDME model efficiently. By separating synchronzation threads from normal threads, the multithreaded processing unit can concentrate on proceeding with computation threads, while the separate synchronization processing unit handles synchronization threads, which are often small. The proposed processor architecture differs also from other related multithreaded processors in dealing with static and dynamic dependencies (operation latencies) between instructions. All static dependency information is integrated in the instruction format, which aims at identifying the possible dependency once an instruction is fetched from the program memory. By splitting all operations with long or dynamic operation latency, the processor pipeline does not need to handle complicated hazard situations occuring in conventional pipelined processors. The multithreaded pipeline sustains its peak performance by exploiting the intra-thread parallel as well as the inter-thread parallelism. Furthermore, by employing the well known I-structure memory, it is expected to handle the data structure access and support synchronization at the data element level efficiently.

Chapter 7

Evaluation of LDME Codes

In this chapter we will first illustrate how a program may be written on a LDME processor. Then, we will show the performance of the LDME code by comparing the LDME code with the conventional sequential code for a pipelined processor with the same basic RISC-instructions as the LDME processor.

As an example, two version of a LDME program for computing the inner-product of the two floating-point vectors have been written: for the first one it is assumed that the two input vectors reside in the local data memory, for the second it is assumed that the two input vectors reside in remote memory modules, which are accessed only by split-phase load/store instructions. The C program of the inner product is as follows:

```
sum = 0;
for (i = 0; i < N; i++ )
     sum = sum + A[i] * B[i];
```

7.1 The Conventional Sequential Code

The associated execution context for the sequential code contains the following data, which is accessed relative to its base address

N : Loop constant
AI: Pointer to A(0)
BI: Pointer to B(0)
SUM: Result

Below is the sequential code written using the RISC-instructions defined in the last section. It is also assumed that all floating point instructions are executed on functional units with the same execution time as in the LDME processor, but not split.

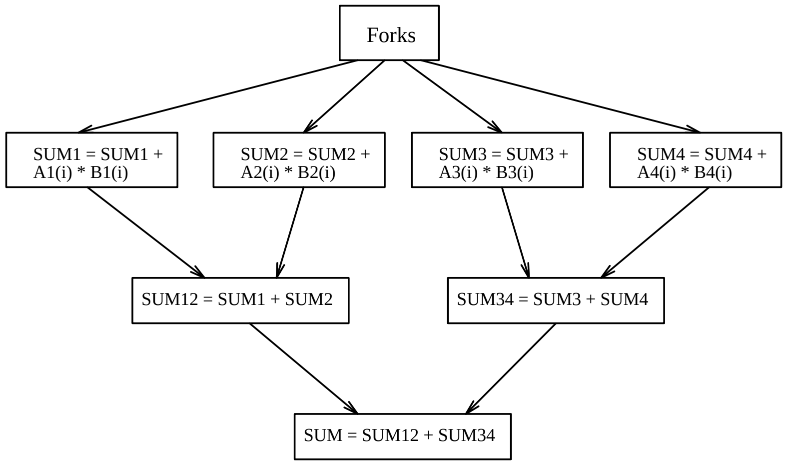


Figure 7.1. A partitioning of the inner product algorithm

```
      load Ra, AI           -- load pointer to A
      load Rb, BI           -- load pointer to B
      load Rn, N            -- load loop constant
      load Rsum, #0         -- initiate register Rsum

Loop:
      load RaI, Ra          -- load A(i) into RaI
      load RbI, Rb          -- load B(i) into RbI
      fmul Rc, RaI, RbI     -- Rc = A(i) * B(i)
      fadd Rsum, Rsum, Rc   -- Rsum = Rsum + Rc
      add  Ra, Ra, #1       -- increase Ra
      add  Rb, Rb, #1       -- increase Rb
      sub  Rn, Rn, #1       -- decrease Rn
      gtp  Rt, Rn, #0       -- Rt = Rn > 0
      tjump Rt, Loop        -- jump if true

      store SUM, Rsum       -- store result
```

7.2 The LDME Code

To write the corresponding LDME code the whole computation should be partitioned into several smaller subcomputations. To simplify simulation of the program, it is assumed that the whole computation is constructed in a single code block, no dynamic loop unrolling is considered, and the corresponding activation is allocated to a simple processing element[1]. Figure 7.1 shows a partitioning of the computation into four smaller subcomputations, which are concurrently executed and then synchronized to sum the results generated by the subcomputations.

[1]This assumption avoids the interference of the run time system (e.g. the resource manager), which will otherwise be called at dynamic loop unrolling.

Also different from the sequential code, all floating point operations are split. This means, as seen in the previous section, that corresponding synchronization threads are initiated for saving the results and initiating further threads. Since the synchronization processing unit operates asynchronously with the multithreaded processing unit, synchronization threads are executed in parallel with the normal threads in the multithreaded processing unit.

For the partitioning in Figure 7.1 the following data frame is necessary for the LDME code that computes the inner-product of two local vectors:

n = N/4 : Loop constant for subcomputation
AI1: Pointer to A1(0)
BI1: Pointer to B1(0)
C1: Temporary variable for A1(i)*B1(i)
Sum1, Sum12: Temporary variable
m1: Loop variable
AI2: Pointer to A2(0)
BI2: Pointer to B2(0)
C2: Temporary variable for A2(i)*B2(i)
Sum2, Sum34 : Temporary variable
m2: Loop variable
AI3: Pointer to A3(0)
BI3: Pointer to B3(0)
C3: Temporary variable for A3(i)*B3(i)
Sum3, Sum : Temporary variable/result
m3: Loop variable
AI4: Pointer to A4(0)
BI4: Pointer to B4(0)
C4: Temporary variable for A4(i)*B4(i)
Sum4 : Temporary variable
m4: Loop variable

For the LDME code that computes the inner-product of two remote vectors, two extra locations in the data frame are necessary for temporarily saving the fetched data values from the remote memory modules.

The program graph for both cases is shown in Figure 7.2, where the dashed lines represent split transaction arcs.

For the first case, in which both input vectors reside in the local data memory. The main thread for initiating four subcomputations, the threads for one subcomputation, the corresponding synchronization threads, and the threads for generating the final result are defined as follows:

```
main:  fork sub1       -- initiate the first subcomputation
       fork sub2       -- initiate the second subcomputation
       fork sub3       -- initiate the third subcomputation
       fork sub4       -- initiate the four subcomputation
```

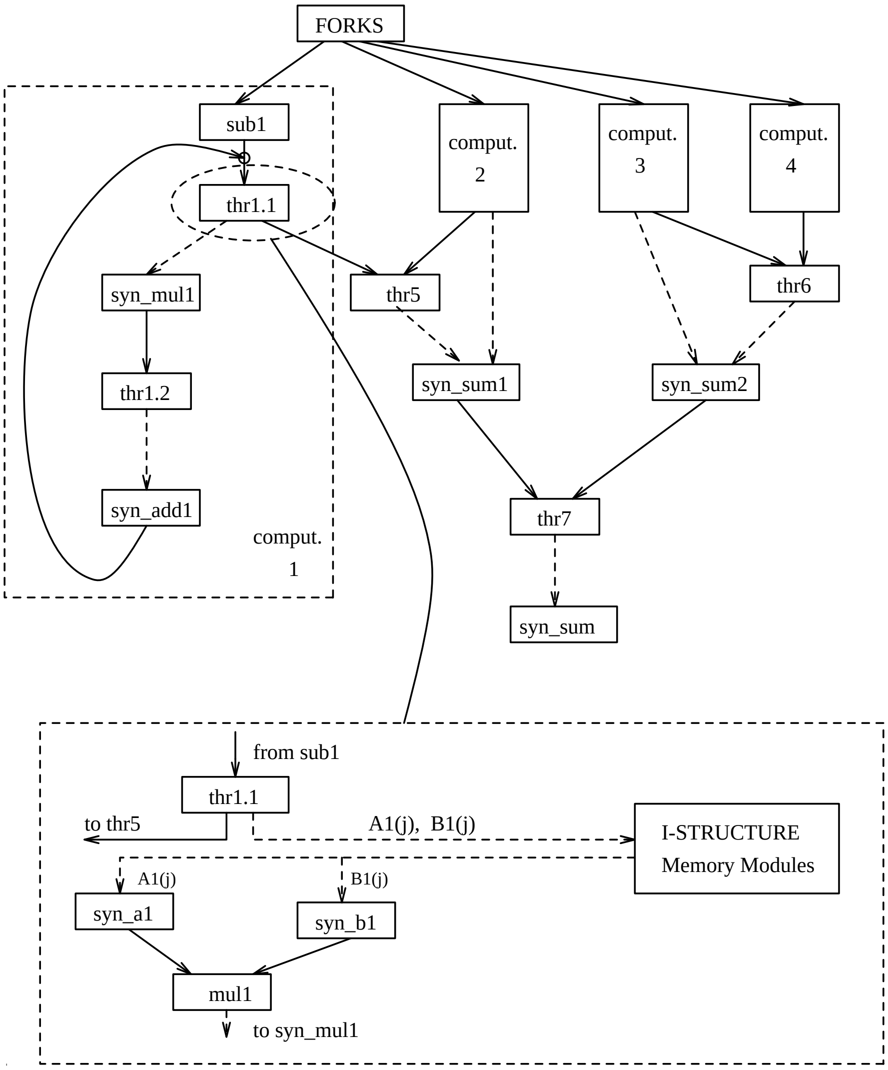


Figure 7.2. The program graph of the inner-product algorithm

```
----------- threads for the first subcomputation ---------------

sub1:                  -- the first subcompuation
       load Ga, A1     -- load pointer Ai to a global register
       load Gb, B1     -- load pointer Bi to a global register
       load Rm, n      -- load the loop constant N/4
       store m1, Rm    -- store the loop constant in m1
       fork thr1.1     -- initiate thr1

thr1.1:
       load Rn, m1     -- load loop constant
       gtp  Rt,Rn,#0   -- true if Rn = 0
       tfork Rt,thr5   -- initiate thr5 if true
       tjump Rt, end
       load RaI, Ga    -- load A1(j)
       load RbI, Gb    -- load B1(j)
       fmul RaI, RbI, Syn_mul1.1
                       -- A1(j)*B1(j), initiate Syn_mul1
       add Ga,Ga,#1    -- increase Ga
       add Gb,Gb,#1    -- increase Gb
       sub Rm, Rm, #1  -- decrease Rn
  end: store m1, Rm    -- store Rm back

thr1.2:
       load Rc, C1     -- load A1(j)*B1(j)
       load Rs, Sum1   -- load Sum1
       fadd Rs, Rc, Syn_add
                       -- Rs + Rc, initiate Syn_add1


------------- threads for producing final result  ------------

thr5(2):
       load Rs1,Sum1   -- load the result of 1st comp.
       load Rs2,Sum2   -- load the result of 2th comp.
       fadd Rs1,Rs2,Syn_sum1
                       -- sum1 + sum2, initiate Syn_sum1

thr6(2):
       load Rs1,Sum3   -- load the result of 3th comp.
       load Rs2,Sum4   -- load the result of 4th comp.
       fadd Rs1,Rs2,Syn_sum2
                       -- sum3 + sum4, initiate Syn_sum2
thr7(2):
       load Rs1,Sum12  -- load the result of 1-2th comp.
       load Rs2,Sum34  -- load the result of 3-4th comp.
       fadd Rs1,Rs2,Syn_sum3
                       -- sum12 + sum34, initiate Syn_sum

thr8:
       ...

---------------- latency synchronization threads ---------------

Syn_mul1:
       rstore V, C1    -- save V-register into C1
```

```
       start thr1.2    -- initiate thr1.2
       stop

Syn_add1:
       rstore V, Sum1  -- save V-register into Sum1
       start thr1.1    -- initiate thr1.1
       stop

Syn_sum1:
      rstore V, Sum12  -- save V-register into Sum12
      start thr7       -- initiate thr5
      stop

Syn_sum2:
      rstore V, Sum34  -- save V-register into Sum12
      start thr7       -- initiate thr5
      stop


Syn_sum:
      rstore V, Sum    -- save V-register into Sum
      start thr8       -- initiate thr6
      stop
```

Note that the numbers in the parentheses after the thread names are the synchronization count of the corresponding threads.

As described in section 6.2.2, a global register set with 8 registers is allocated to each resident activation. Thus, in the LDME code above, two global registers are assigned to each subcomputation, i.e. Ga, Gb, which are used to save two pointers to two sub-vectors Ai and Bi. Other variables such as the loop variable m_i must be first loaded and then saved before the thread $thr1.1$ terminates.

For the second case, in which both input vectors reside in the remote memory modules, two load instructions for loading $Ai(j)$ and $Bi(j)$ in the thread $thr1.1$ must be split. Two extra temporary locations $AiandBi$ in the data frame are introduced for each subcompution to save the fetched data values from the remote memory module. Thus, the thread *thr1.1* above may be rewritten as follows:

```
thr1.1:
       load Rn, mi       -- load loop constant
       gtp  Rt,Rn,#0     -- true if Rn = 0
       tfork Rt,thr5     -- initiate thr5 if true
       tjump Rt, end
       Iread Ga, syn_ai  -- read Ai(j) from I-structure
       lread Gb, syn_bi  -- read Bi(j) from I-structure
       add Ga,Ga,#1      -- increase Ga
       add Gb,Gb,#1      -- increase Gb
       sub Rm,Rm,#1      -- decrease Rn
  end: store mi, Rm      -- store Rm back
```

```
mul1(2):
        load RaI,A1       -- load A1(j)
        load RbI,B1       -- load B1(j)
        fmul RaI, RbI, Syn_mul1.1
                          -- A1(j)*B1(j), initiate Syn_mul

syn_a1:
        rstore V, A1      -- save V-register into A1
        start mul1        -- initiate syn_ld
        stop

syn_b1:
        rstore V, B1      -- save V-register into B1
        start mul1        -- initiate syn_ld
        stop
```

Surely, the LDME code looks more complicated than the conventional sequential code due to more smaller threads, which are executed *data driven*, and due to more load/store instructions, which are used to load and store some temporary values. The key advantage of the LDME code, which should be obvious provided the associated architecture is kept in mind, is that all instructions executed in the multithreaded pipeline are nonblocking. This leads to a decisive performance advantage: *the processor sustains its peak performance provided enough parallelism exists in the program.* Furthermore, all synchronization threads are executed on a separate synchronization processing unit, which reduces the burden of the multithreaded processing unit.

7.3 Comparison and Analysis

7.3.1 Execution Latency of Floating Point Operations

Before comparing the LDME code with the corresponding sequential code, let us first examine the execution latencies of floating point operations, which influence the performance of the sequential code as well as the LDME code. Generally, floating point operations are performed in multiple cycles. Therefore, two floating point operations $fadd$ and $fmul$ are split in the LDME code. In order to compare the performance of the LDME code with conventional sequential code given in the last section, the execution latencies of floating point operations on some recently announced important RISC microprocessors are listed in Table 7.1:

Processors	Execution Latencies (cycle)					
	Float Add		Float Multiply		Float Divide	
	single	double	single	double	single	double
DLX(Mips R2010) [Hen 90]	2	3	4	6	12	27
Alpha(21064) [DEC 92a]	6	6	6	6	34	63
Mips R4000 [Mil 92]	4	4	7	8	23	36
M88110 [Die 92]	3	3	3	3	13	N.A.
PA-RISC1.1 [Del 92]	2	2	2	2	8	15
SuperSPARC [Bla 92]	3	3	3	3	6	7

Table 7.1: Typical execution latencies of floating point operations

As shown, for single precision floating point operations, addition has an execution latency between 2 and 6 cycles, multiplication an execution latency between 2 and 7, and the division an execution latency between 6 and 34 cycles. For double precision operations, the execution latencies are longer. We do not list the execution latencies for the integer multiplication and division here. It should be noted, however, that integer multiplication and division may have longer execution latencies dependent on algorithms used in implementation. For example, the Mips R4000 uses a 2-bit Booth algorithm for integer multiply, which results in an execution latency of 10 cycles for the 32 bit multiply operation and of 20 cycles for the 64 bit multiply operation. For integer divide, Mips R4000 uses a 1-bit-per-iteration, nonrestoring algorithm, which results in an execution latency of 69 cycles for the 32 bit divide operation and of 133 cycles for the 64 bit divide operation [Mil 92].

Both the sequential code and the LDME code have been executed in the LDME processor simulator in which the floating point adder, the multiplier and the synchronization unit are pipelined. In the following the experimental results for these two cases are presented.

7.3.2 Local Input Vectors

For the first case, both input vectors are stored in the local data memory. Therefore, no extra memory latency occurs in fetching the input vectors. The performance of the codes (the sequential code as well as the LDME code) depends mainly on the pipeline lengths of the floating point adder and multiplier. The synchronization processing unit has a fixed operation rate. In the example of the inner-product the synchronization processing unit completes every latency synchronization thread in three clock cycles.

In the LDME code we have also seen a clear and unpleasant consequence of our choice in splitting floating point operations: the code size of each subcomputation is larger than in the sequential code, because each split operation will initiates a latency synchronization thread, which further initiates a normal thread containing three instructions (e.g. *thr1.2*) in order to continue the next floating point operation (see the LDME code). The execution of the

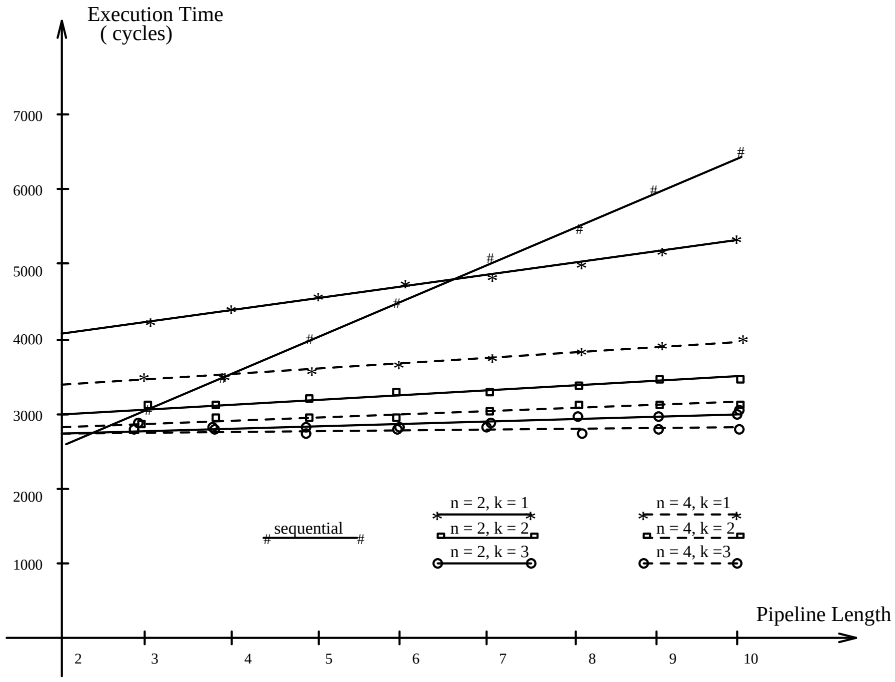


Figure 7.3. The performance of the inner-product with local input vectors

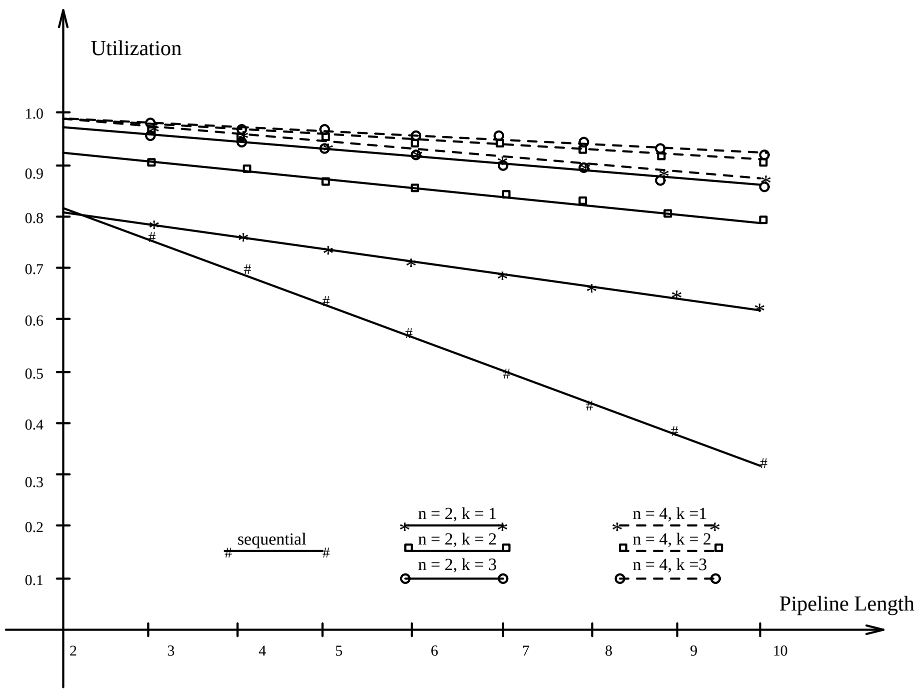


Figure 7.4. The utilization of the inner-product with local input vectors

synchronization thread may be overlapped with other enabled normal threads. This extra normal thread is, however, the associated overhead in comparison to its sequential code. The solution to avoid those extra normal threads is to *group* together several split operations. Thus, instead each split operation initiates a latency synchronization thread and a normal thread, several split operations together initiate a latency synchronization thread and a normal thread. To support grouping of split operations, a direct store mode has been introduced for all local split-phase instructions (see the instruction set of the last chapter).

To accomplish the grouping, the loop must be unrolled. For the LDME code of the inner product, the second normal thread of each subcomputation, e.g. *thr1.1*, must be changed so that only the last split operation will initiate the corresponding latency synchronization thread.

The simulation results in Figure 7.3 and 7.4 show the performance of different codes and their corresponding multithreaded pipeline utilization. It was assumed that each input vector contains 240 elements, i.e. $N = 240$. The whole computation was divided into 2 and 4 subcomputations, i.e. $n = 2, 4$. k is the unrolled times of the subcomputation. Further, it was assumed that floating point adder and mulitplier have the same pipeline length, which varies between 2 and 10 pipeline stages.

As expected, the LDME codes have a better performance when the pipeline length of floating point adder and multiplier is increased. To further improve LDME codes, grouping split operations by unrolling loop is necessary as shown in Figure 7.3. Another interesting result shown in Figure 7.3 is that the performance of the LDME code is **less sensitive** to the pipeline lengths of the floating point operations than the sequential code for conventional pipelined processors. The reason for this result is that the execution of floating point operations are overlapped with other enabled normal threads. If enough parallelism exists in a program, the operation latency of split operations can be tolerated completely. This can also be observed in Figure 7.4, where the LDME codes have an obviously better utilization than the sequential code. An architectural advantage of this property of the LDME code is to encourage the design of fast funtional units with deep pipelines.

7.3.3 Remote Input Vectors

For the second case, both input vectors reside in remote memory blocks. Hence, extra memory latency occurs in accessing the input vectors. Differently from the first case, the influence of the pipeline length of the floating point adder and multiplier on the processor performance becomes only a small fraction due to long memory latency. Therefore, here it was assumed that floating point adder and multiplier have the same fixed execution latency (5 cycles in our experiments). Four different memory latencies, which can be used to represent four different

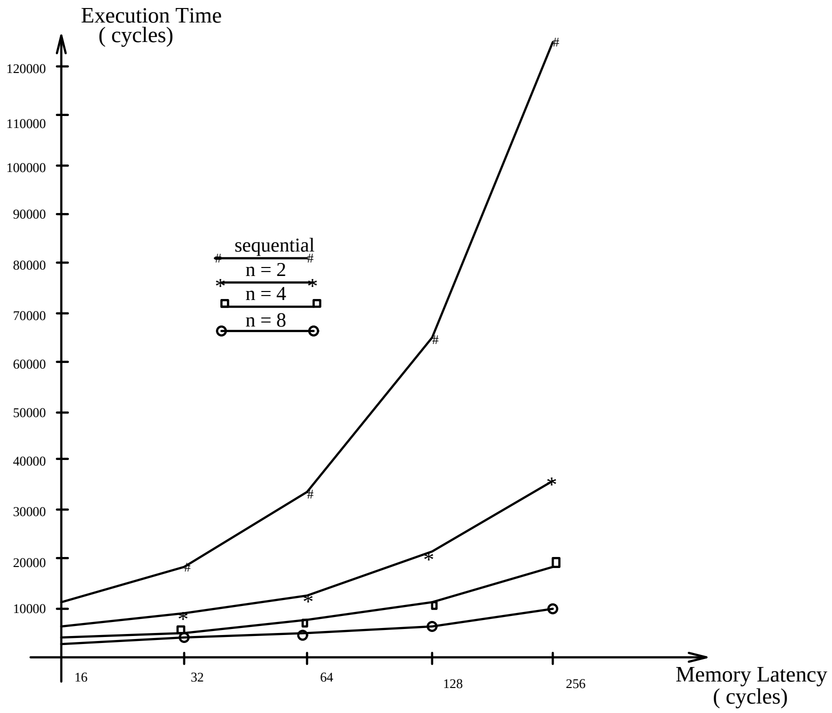


Figure 7.5. The performance of the inner-product with remote input vectors

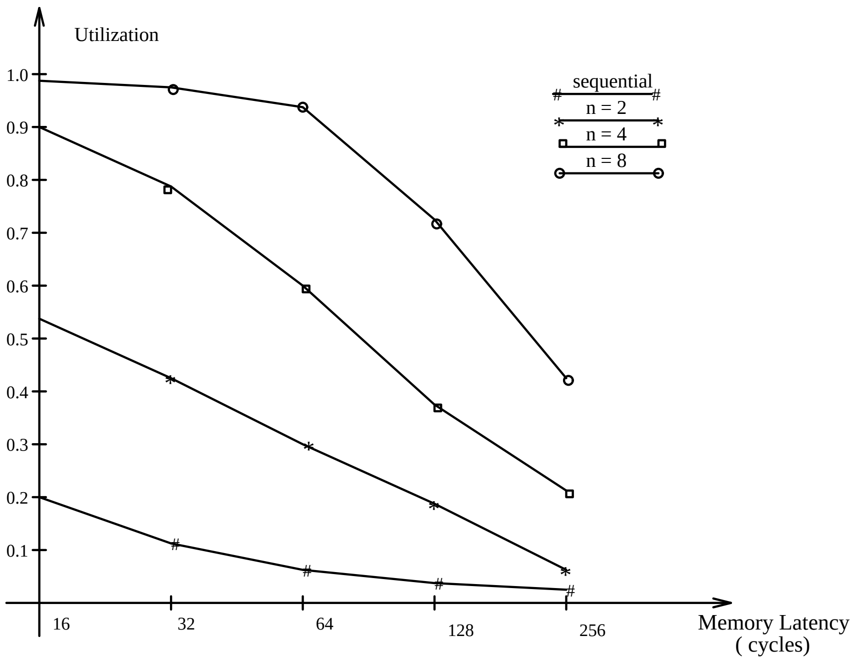


Figure 7.6. The utilization of the inner-product with remote input vectors

multiprocessor configurations, have been used in our experiments.

The simulation results in Figure 7.5 and 7.6 show the performance of different variation of the code and the corresponding multithreaded pipeline utilization. As in the first case, each input vector contains 240 elements. No loop unrolling was done because of the small influence of local split operations on the processor performance in a multiprocessor environment.

The simulation results clearly show the performance advantage of the LDME code in comparison to the conventional sequential code. Three different partitionings of the computions are examined: n = 2, 4, and 8. For example, in the multiprocessor configuration with a memory latency of 64 cycles, the partitioning of the computation into 8 subcomputations reaches a processor utilization of 0.94. For the partitioning of computation into 4 subcomputation, the processor utilization is dropped to only 0.59. For a longer memory latency more parallelism has to be exploited in order to keep the processor busy.

7.4 Summary and Remark

This chapter has demonstrated the benefits of the LDME codes in comparison to the conventional sequential code. The simulation results for the first case (local input vectors) also showed the unpleasant consequence of splitting each floating point operation due to some extra overhead in synchronizing further operations. The LDME processor supports grouping of several split operations in order to reduce such extra overhead. This technique has led to an obvious performance improvement. The smaller sensitivity of the pipeline lengths is another novel character of the LDME code, which is of importance in building fast pipelined functional units. The LDME codes showed their clear performance advantage in multiprocessor environments. The simulation results presented in this chapter are very close to the performance analysis presented in chapter 4.

It should be noted, however, that the grouping of split operations assumes that the ordering of storing the results of split operations should not be changed. In order to reach the sequential consistency, the processor must guarantees that all issued split operations have completed before the last split operation initiates its latency synchronization thread. The sequential consistency for local split operations can easily supported by the hardware as well as by the compiler. For example, the central split operation window in the multithreaded processing unit can register the time when a split operation enters the window. By issuing a split operation to the corresponding functional unit the central split operation window selects an operation for issuing according to its enter time. The compiler can rearrange the instruction sequence so that the last split operation will be executed after other issued split operations have surely been completed. In a multiprocessor environment, it may be unrealistic to assume the sequential consistency as in [Boo 92], because the time when remote accesses return may depend on many factors such as path distance, network conflicts, etc.

Chapter 8

Conclusion

The lessons learned over the last years for the design of von Neumann machines and dataflow machines have led to the belief that some fundamental changes in both architectural models are required to build more powerful and efficient computer systems. Even though the rapid development of VLSI technology has made it possible to build fast single-chip processors including high-speed floating-point arithmetic (e.g. DEC's 21064 (Alpha), TI's SuperSparc, MIPS's R4000, etc.), no fundamental changes in the processor architectures have been taken. Furthermore, some problems like pipeline hazards have become more serious due to multiple instruction issues per cycle in superscalar processors or due to longer pipelines in superpipelined processors. On the other hand, the recent research on dataflow machines has proven that dataflow machines can solve the two fundamental problems, namely latency and synchronization, in building any scalable multiprocessor, but their inherent weaknesses both in the execution model and in the associated hardware requirements have prevented dataflow machines from wide acceptance until now. Moreover, little attention has been paid to the performance of the processing elements (uniprocessors) in a multiprocessor environment, which, however, surely influences the performance of the overall system.

The overall goal of this work has been to understand the performance advantages of an alternative – multithreading – both to conventional pipelining and dataflow model. As discussed in this thesis, multithreaded architectures combine architectural features of von Neumann and conventional dataflow models, avoiding the inefficiencies of both models. Starting with the observation that conventional pipelined processors suffer from performance loss due to pipeline hazards in uniprocessors and due to two fundamental problems as building blocks in multiprocessors and with the fact that both problems are unavoidable in machines based on von Neumann model. We have studied performance and architectural advantages of multithreaded execution in this work. The performance analysis of various multithreaded models has shown that a multithreaded processor with the SORLD strategy can sustain its peak performance even with only a small number of active threads in the processor. By exploiting both intra-thread and inter-thread parallelism, the proposed multithreaded execution model – latency-directed multithreaded execution model (LDME) – and its associated architectural support (a multithreaded processor with the SORLD strategy) take the advantages of multithreading to hide pipeline hazards in a uniprocessor environment as well as long unpredicatable memory and synchronization latencies in a multiprocessor environment.

The main departure of the proposed multithreaded processor from conventional pipelined processors and other multithreaded processors lies in processing latency synchronization threads and normal threads separately. Thus, all latency sensitive operations can be rephased as split transactions which separately initiate an operation and later explicitly synchronize on the availability of the result by a short latency synchronization thread. This leads to easy handling of different pipeline hazards as well as long unpredicatable memory or synchronization latencies. As described, the proposed multithreaded processor requires only simple hardware support, because no special hardware mechanisms for detecting and resolving the variety of pipeline hazards, no presence-bits in each location of data frame memory and associated control logic, etc. are required. The processor can sustain peak performance if a relative small amount of parallelism exists in a programs.

Of course, it is impossible for this work to deal with all possible aspects of the multithreaded execution and the associated architectural support. For example, we have not touched software issues such as suitable program representation, language semantics, optimal choice of thread size, synchronization effects on processor performance, resource management at run time, organization of data structure memory, interconnection network structure, etc.. It is claimed, however, that solutions applicable to other machines such as dataflow machines or other multithreaded machines are equally applicable to our multithreaded processor because of the relevant architectural similarities in storage demand, register requirements, procedure call/return mechanism, etc.

Designing a new processor is a complex task, which includes a variety of different subtasks. The results presented in this thesis sets the perspective for further research:

- **Detailed simulation:** The proposed multithreaded processor has been simulated at the instruction level in C. A more detailed simulation of the architecture behaviour at the register transfer level can lead to more accurate estimation of processor performance. It is believed that VHDL is a suitable description language for describing our multithreaded processor (especially for describing the multithreaded pipeline as seen in chapter 6).

- **LDME code generation:** More application programs should be written with the LDME model in order to estimate the code efficiency and evaluate the instruction set. In chapter 6 only basic instructions for the LDME model are included; more instructions, especially those instructions supporting efficient resource management and program organization may be necessary.

- **Synchronization effects:** Synchronization effects on processor performance are difficult to determine statically, because the synchronization latency is generally unpredictable. In our study, the synchronization effects have been minimized by the assumption that a program has a small fork/join phase at the beginning and the end of program execution and a fixed number of threads during the main execution period. This assumption may not be true for programs written in some modern parallel languages like functional or logic languages.

- **Register management and size:** We have chosen 8 local register sets and two global register sets. Each register set contains 8 registers, which is, however, only an empirical number extracted from the experience of MASA [Hal 88]. As shown in the LDME program for the inner product in the previous chapter, it is desirable to enlarge the two global register sets, which can reduce the frequency of *load/store* instructions in normal threads. The optimal size of register sets can only be determined through practical experience with different LDME codes.

- **VLSI implementation:** Once the proposed multithreaded pipeline is described in VHDL at the register transfer level, it is possible to synthesize the multithreaded pipeline with some high level synthese tool. A variety of VLSI design tools including the synthesis tool **Synopsis** are available in the institute (AB TECH, Department of Computer Science, University of Hamburg) where this work was done. Thus, a rapid hardware estimation of the multithreaded pipeline can be achieved by using high level hardware synthesis.

There are also many other similar efforts aimed at combining von Neumann and dataflow architectures. Most of such efforts have been stimulated by the fact that dataflow machines based on pure data flow execution model pioneered by Arvind and Dennis over the last 20 years are inefficient and difficult to build, just as pointed out by Iannucci [Ian 90]:

> *Neither von Neumann architecture nor dataflow architecture alone can achieve the goal of scalable, general purpose parallel computing.*
>
> *With the hybrid perspective, there is great hope.*

This is also our belief as a conclusion from this work.

Bibliography

[Ack 82] W.B. Ackerman. Data Flow Language. *Computer*, 15(2):15–25, 1982.

[Ada 91] R. G. Adams and G. B. Steven. A Parallel Pipelined Processor with Conditional Instruction Execution . *Computer Architecture News*, 19:579–587, 1991.

[Aga 90] A. Agarwal, Beng-Hong Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–115. IEEE, 1990.

[Aga 89] A. Agarwal. Performance Tradeoffs in Multithreaded Processors. MIT VLSI Memo 89-566, MIT Laboratory for Computer Science, Cambridge, MA 02139, 1989.

[Alm 89] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc., 1989.

[Alv 90] R. Alversion et al. The Tera Computer System. In *Proc. of Intl. Conf. on Supercomputing*, pages 1–6, June 1990.

[Ama 84] M. Amamiya and R. Hasegawa. Dataflow Computing and Eager and Lazy Evaluations. *New Generation Computing*, 2(2):105–129, 1984.

[Ama 89] M. Amamiya and R. Taniguchi. An Ultra-Multiprocessing Machine Architecture for Efficient Parallel Execution of Fuctional Languages. *Lecture Notes in Computer Science*, 491:257–281, 1989.

[And 67] D. W. Anderson, F. J. Sparacio and R. M. Tomasulo. The IBM 360 Model 91: Machine Philosophy and Instruction Handling. *IBM J. of Research and Development*, 11(1):8–24, 1967.

[Arv 82] Arvind and K. P. Gostelow. The U-Interpreter. *Computer*, 15(2):42–49, 1982.

[Arv 83] Arvind and R. A. Iannucci. A Critique of Multiprocessing von Neumann Style. In *Proc. of 10th Annual Intl. Symp. on Computer Architecture*, pages 426–436, 1983.

[Arv 87] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *Lecture Notes in Computer Science*, 295:61–88, 1987.

[Arv 86] Arvind and D. E. Culler. Dataflow Architectures. Technical report, MIT, 1986. MIT/LCS/TM-294.

[Arv 88] Arvind, D. E. Culler, and K. Ekanadham. The Price of Asychnonous Parallelsim: an Analysis of Dataflow Architectures. In *Proc. of CONPAR 88, Stream B*, pages 168–182, 1988.

[Arv 90] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. on Computers*, c-39(3):300 – 318, 1990.

[Bab 82] R. G. Babb II. Data-driven Implementation of Data Flow Diagrams. In *Proc. of 6th. Intl. Conf. on Software Engineering*, pages 309–318, 1982.

[Bab 84] R. G. Babb II. Parallel Processing with Large-Grain Data Flow Techniques. *Computer*, 17(7):55–61, 1984.

[Bac 76] J. Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communication of the ACM*, 21(8):613 – 641, 1978.

[Bar 87] S. Barkhordarian. Ramps: A Realtime Structured Small-Scale Data Flow System for Parallel Processing. In *Proc. of Intl. Conf. on Parallel Processing*, pages 610–613, 1987.

[Bec 90] M. Beck and K. Pingali. Static Scheduling for Dynamic Dataflow Machines. *Journal of Paralllel and Distributed Computing*, 10(4):279–288, 1990.

[Bic 87] L. Bic. A Process-oriented Model for Efficient Execution of Data Flow Programms. In *Proc. of 7th. Intl. Conf. on Distributed Computing*, pages 1–2, 1987.

[Bla 92] G. Blanck and S. Krueger. The SuperSPARC Microprocessor. In *Proceeding of COMPCON 92*, pages 136–141, 1992.

[Boo 92] B. Boothe and A. Ranade. Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessors. In *Proc. of 18th Annual Intl. Symp. on Computer Architecture*, pages 214–223, 1992.

[Bue 86] R. Buehrer and K. Ekanadham. Dataflow Principles in Multi-Processor Systems. Technical Report RC 12190, IBM, IBM T.J. Watson Research Center Yorktown Heights, New York, July 1986.

[Bue 87] R. Buehrer and K. Ekanadham. Incorporating Data Flow Ideas into von Neumann Processors for Parallel Execution. *IEEE Trans. on Computers*, c-36(12):1515–1522, 1987.

[But 86] S. Butner and C. A. Staley. A RISC Multiprocessor Based on Circulating Context. In *IEEE Phoenix Conference on Computers and Communications*, March 1986.

[Cha 81] A. E. Charlesworth. An Approach to Scientific Arrary Processing: The Architectural Design of the AP-120B/FSP-164 Family. *Computer*, 14(9):18–27, Sept. 1981.

[Che 71] T. C. Chen. Parallelism, Pipelining and Computer Efficiency. *Computer Design*, 10(1):69–74, Jan. 1971.

[Che 90] Ding-Kai Chen, Hong-Hen Su, and Pen-Chung Yew. The Impact of Synchronization and Granularity in Parallel Systems. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 239–248, 1990.

[Chi 91] T. Chiueh. Multi-Threaded Vectorization. In *The 18th Annual International Symposium on Computer Architecture*, pages 352–361, 1991.

[Coh 89] R. Cohn, et al. Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor. In *Proceedings of the 3th International Conf. on Architectural Support for Programming Language and Operating Systems*, pages 2–14, April 1989.

[Cul 88] D. E. Culler and Arvind. Resource Requirements of Dataflow Programms. In *Proc. of 15th Annual Intl. Symp. on Computer Architecture*, pages 141–150, 1988.

[Cul 90] D. E. Culler. Managing Parallelsim and Resources in Scientific Dataflow Programs. Computation Structures Group Memo 446, MIT Laboratory for Computer Science, Cambridge, MA 02139, March 1990.

[Cul 91a] D. E. Culler et al. Fine-grain Parallelsim with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Fifth Intf. Conf. on Arcitectural Supports for Programming Lauguages and Operating Systems*, pages 164–175, 1991.

[Cul 91b] D. E. Culler and G. M. Papadopoulos. The Explicit Token Store. *Journal of Parallel and Distributed Computing*, 10(4):289–308, January 1991. (Also published as Computation Structures Group Memo 312).

[Dai 88] K. C. Dai. Large-Grain Dataflow Computation and Its Architectural Support. Ph.D thesis, Technical University of Berlin, West Germany, 1988.

[Dai 90] K. C. Dai and K. Giloi. A Basic Architectural Supporting LGDG Computation . In *Proc. of Intl. Conf. on Supercomputing*, pages 23–33, 1990.

[Dai 91] K. C. Dai and K. Giloi. Language Styles for Programming Multiprocessing - Imperative, Functional or Beyond? . In *Proc. of the Fourth ISMM/IASTED Intl. Conference: Parallel and Distributed Computing and Systems*, pages 425–432, October 1991.

[Dal 86] W. J. Dally et al. Architecture of a Message-Driven Processor. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 189–196. IEEE, 1986.

[Dal 87] W. J. Dally. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Publishers, 1987.

[Dal 89] W. J. Dally and D. Scott Wills. Universal Mechanisms for Concurrency. In *PARLE 89: Parallel Architectures and Languages Europe, Volume 1: Parallel Architectures*, pages 19–33, 1989.

[Dal 92] W. J. Dally et al. The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro*, 12(2):23–39, April 1992.

[Das 89] S. Dasgupta. *Computer Architecture - A Modern Synthesis, Vol. 1: Foundations.* John Wiley & Sons, Inc. 1989.

[Dav 82] A. L. Davis and R. M. Keller. Data Flow Program Graph. *Computer*, 15(2):26–41, 1982.

[DEC 92a] *DECChip 21064-AA RISC Microprocessor - Preliminary Data Sheet.* Digital Equipment Corporation, Maynard, Massachusetts, 1992.

[DEC 92b] *Alpha Architecture Handbook.* Digital Equipment Corporation, 1992.

[Del 92] E. DeLano et al. A High Speed Superscalar PA-RISC Processor. In *Proceeding of COMPCON 92*, pages 116–121, 1992.

[Den 80] J. B. Dennis. Data Flow Supercomputer. *Computer*, 13(11):48–56, 1980.

[Den 85] J. B. Dennis. The Data Flow Computation. In M. Broy, editor, *Control Flow und Data Flow: Concepts of Distributed Programming*, pages 346–398. Springer-Verlag, 1985.

[Den 84] J. B. Dennis, G. R. Gao, and K. W. Todd. Modelling the Weather with a Data Flow Supercomputer. *IEEE Trans. on Computers*, C-33(7):592–603, 1984.

[Der 87] J. A. DeRosa and H. M. Levy. An Evaluation of Branch Architectures. In *Proc. of 14th Annual Intl. Symp. on Computer Architecture*, pages 10–16, 1987.

[Die 92] K. Diefendorff and M. Allen. Organization of the Motorola 88110 Superscalar RISC Microprocessor. *IEEE Micro*, 12(2):41–63, April 1992.

[Din 89] A. Dinning. A Survey of Synchronization Methods for Parallel Computers. *Computer*, 22(7):66–77, July 1989.

[Eka 86] K. Ekanadham. Multi-Tasking on a Dataflow-like Architecture. Technical Report RC 12307, IBM, IBM T.J. Watson Research Center Yorktown Heights, New York, Nov. 1986.

[Fan 91] X. Fan. Queue-based Primitives for a Multithreaded Architecture. *Microprocessing and Microprogramming*, 34(1992):113–116, EUROMICRO 91, Vienna, Austria, Sept. 1991.

[Fan 92a] X. Fan. Realization of Multiprocessing on a RISC-like Architecture. *Microprocessing and Microprogramming*, 33(1991/92):195–206, 1992.

[Fan 92b] X. Fan. Analysis of Multithreaded Architectures - A Case Study. *Microprocessing and Microprogramming*, 37(1993):87–90, EUROMICRO 92, Paris, France, Sept. 1992.

[Far 91] M. K. Farrens and A. R. Pleszkun. Strategies for Archieving Improved Processor Throughput. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 362–369, 1991.

[Fla 89] H. P. Flatt. Performance of Parallel Processors. *Parallel Computing*, 12:1–20, 1989.

[Fly 72] M. J. Flynn and A. Podvin. Shared Resource Multiprocessing. *Computer*, 5(3):20–28, March 1972.

[Gaj 82] D. D. Gajski et al. A Second Opinion on Data Flow Machines and Languages. *Computer*, 15(2):26–41, 1982.

[Gaj 85] D. D. Gajski and Jih-Kwon Peir. Essential Issues in Multiprocessor Systems. *Computer*, 18(6):9–27, June 1985.

[Gau 86] J. L. Gaudiot. Structure Handling in Data-Flow Systems. *IEEE Trans. on Computers*, C-35(6):489–501, 1986.

[Gau 85] J. L. Gaudiot et al. A Distributed VLSI Architectures for efficient Signal and Data Processing. *IEEE Trans. on Computers*, C-34(12):1072–1087, 1985.

[Gau 87] J. L. Gaudiot and L. T. Lee. Multiprocessor Systems Programming in a High-level Dataflow Language. *Lecture Notes in Computer Science, No. 258*, pages 134–151, 1987.

[Gil 81] W. K. Giloi. *Rechnerarchitektur*. Springer-Verlag, Berlin, Heidelberg und New York, 1981.

[Gra 91] V. G. Grafe and J. E. Hoch. The EPSILON Multiprocessor System. *Journal of Parallel and Distributed Computing*, 10(4):309–318, January 1991.

[Gro 88] T. R. Gross et al. Measurement and Evaluation of the MIPS Architecture and Processor. *ACM Transaction on Computer Systems*, 6(3):220–259, August 1988.

[Gup 91] A. Gupta et al. Comparative Evaluation of Latency and Tolerating Techniques. In *The 18th Annual International Symposium on Computer Architecture*, pages 254–263, 1991.

[Gur 80a] J. Gurd and I. Watson. Data Driven System for High Speed Parallel Computing- Part 1: Structuring Software for Parallel Execution. *Computer Design*, pages 91–100, June 1980.

[Gur 80b] J. Gurd and I. Watson. Data Driven System for High Speed Parallel Computing- Part 2: Hardware Design. *Computer Design*, pages 79–106, July 1980.

[Hal 88] R. H. Halstead and T. Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *Proc. of 15th Annual Intl. Symp. on Computer Architecture*, pages 443–451, 1988.

[Hen 90] J. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann Pub. Inc., 1990.

[Hir 92] H. Hirata et al. Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessors. In *Proc. of 18th Annual Intl. Symp. on Computer Architecture*, pages 214–223, 1992.

[Hoa 78] C. A. R. Hoare. Communicating Sequential Processes. *Communication of the ACM*, 21(8):666–677, 1978.

[Hwa 84] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill Book Company, 1984.

[Ian 88] R. A. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In *Proc. of 15th Annual Intl. Symp. on Computer Architecture*, pages 131–140, 1988.

[Ian 90] R. A. Iannucci. *Parallel Machines: Parallel Machine Languages - The Emergence of Hybrid Dataflow Computer Architectures* . Kluwer Academic Publishers, 1990.

[Kam 79] W. J. Kaminsky and E. S. Davidson. Developing a Multiple-Instruction-Stream Single-Chip Processor. *Computer*, 12(12):66–76, 1979.

[Kec 92] S. W. Keckler and W. J. Dally. Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism. In *Proc. of 18th Annual Intl. Symp. on Computer Architecture*, pages 202–213, 1992.

[Joh 91] M. Johnson. *Superscalar Microprocessor Design.* Prentice Hall, 1991.

[Jor 85] H. F. Jorden. HEP Architecture, Programming and Performance. In J. S. Kowalik, editor, *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, pages 1–40. The MIT Press, 1985.

[Jou 89] N. P. Jouppi and D. W. Wall. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines. In *Third Intl. Conf. on Architectural Supports for Programming Lauguages and Operating Systems*, pages 272–282, 1989.

[Kar 87] A. H. Karp. Programming for Parallelism. *Computer*, 20(3):43–57, 1987.

[Kim 91] S. D. Kim et al. Modeling Overlapped Operation between the Control Unit and Processing Elements in an SIMD Machine. *Journal of Paralllel and Distributed Computing*, 1(12):329–342, 1991.

[Kir 89] C. Kirchner and U. Skupin. Die Puffermaschine: Entwurf und Realisierung einer fehlertoleranten , objektorientierten Rechnerarchitektur, teil 2: Implementation. Technical Report 391, FFM, Forschungsinstitut fuer Funk und Mathematik, D-5307 Wachtberg-Werthhoven, West Germany, Mai 1989. (in German).

[Kog 81] P. M. Kogge. *The Architecture of Pipelined Computers.* Heimisphere Publishing Corporation, 1981.

[Lil 88] D. J. Lilja. Reducing the Branch Penalty in Pipelined Processors. *Computer*, 21(7):47–55, July 1988.

[McC 91] D. C. McCrackin. Eliminating Interlocks in Deeply Pipelined Processors by Delay Enforced Multistreaming. *IEEE Transactions on Computers*, C-40(10):1125–1132, October 1991.

[McF 86] S. McFarling and J. Hennessy. Reducing the Cost of Branch. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 396–403. IEEE, 1986.

[Mey 82] G. J. Meyer. *Advances in Computer Architectures.* John Wiley and Sons, second edition, 1982.

[Mil 74] E. F. Miller. A Multiple-Stream Registerless Shared Resource Processor. *IEEE Trans. on Computers*, C-23:2777–285, May 1974.

[Mil 92] V. M. Milutinovic et al. The Mips R4000 Processor . *IEEE Micro*, 12(2):11–22, April 1992.

[Moh 91] E. Mohr, D. A. Kranz, and R. H. Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[Naj 87] W. Najjar and J. L. Gaudiot. Multi-Level Execution in Data Flow Architecture. In *Proc. of Intl. Conf. on Parallel Processing*, pages 32–37, 1987.

[Nik 89] S. R. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proc. of 16th Annual Intl. Symp. on Computer Architecture*, pages 262–272, 1989.

[Nik 92] R.S. Nikhill, G.M. Papadopoulos and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proc. of 18th Annual Intl. Symp. on Computer Architecture*, pages 156–167, 1992.

[Nut 91] P. R. Nuth and W. J. Dally. A Mechanism for Efficient Context Switching. Proc. of Intl. Conf. on Parallel Processing, pages 301–304, 1991.

[Omo 91] A. R. Omondi. Design of a High Performance Instruction Pipeline. *Computer System Science and Engieering*, pages 13–29, 1991.

[Pap 89] G. M. Papadopoulos. Program Development and Performance Monitoring on the Monsoon Dataflow Multiprocessor. Computation Structures Group Memo 303, MIT Laboratory for Computer Science, Cambridge, MA 02139, October 1989.

[Pap 91a] G. M. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor*. MIT Press, Cambridge, MA, 02141, 1991. (1988 MIT Ph.D. Thesis, also published as MIT LCS TR 432).

[Pap 91b] G. M. Papadopoulos and K. R. Traub. Multithreading: A Revisionist View of Dataflow Architecture. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 342–351. IEEE, 1991.

[Par 91] W. W. Park et al. Performance Advantages of Multithreaded Processors. In *Proc. of Intl. Conf. on Parallel Processing*, pages I97–I101, 1991.

[Pre 85] B. R. Preiss and Hamacher. Data Flow on a Queue Machine. In *Proc. of 12th Annual Intl. Symp. on Computer Architecture*, pages 342–351, 1985.

[Qua 91] D. J. Quammen and R. D. Miller. Flexible Register Management for Sequential Programs. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 320–329, 1991.

[Saa 91] R. H. Saavedra-Barrera and D. E. Culler. An Analytical Solution for a Markov Chain Modeling Multithreaded Execution. Technical Report UCB/CSD91/623, Computer Science Division(EECS), University of California, Berkeley, California 94720, April 1991.

[Saa 90] R. H. Saavedra-Barrera, D. E. Culler, and T. von Eicken. Analysis of Multithreaded Architectures for Parallel Computing. In *2nd Annual ACM Symp. on Parallel Algorithms and Architectures*, pages 169–178, 1990.

[Sat 92] M. Sato et al. Thread-based Programming for the EM-4 Hybrid Dataflow Machine. In *Proc. of 18th Annual Intl. Symp. on Computer Architecture*, pages 146–155, 1992.

[Sei 84] C. L. Seitz. Concurrent VLSI Architectures. *IEEE Transactions on Computers*, C-33(12):1247–1265, December 1984.

[Sev 89] K. C. Sevcik. Charaterizations of Parallelism in Applications and Their Use in Scheduling. In *Intl. Conf. on Measurement and Modeling of Computer Systems*, pages 171–180, May 1989.

[Sha 74] L. E. Shar and E. S. Davidson. A Multimini Processor System Implemented Through Pipelining. *Computer*, pages 42–51, February 1974.

[She 91] J Shelter and S Butner. Multiple Stream Execution on the DART Processor. In *Proc of Intl. Conf. on Parallel Processing*, August 1991.

[Smi 78] B. Smith. A Pipelined, Shared Resource MIMD Computer. In *Proc. of Intl. Conf. on Parallel Processing*, pages 6–8, 1978.

[Smi 81] B. Smith. Architecture and Application of the HEP multiprocessor Computer System. In *Proc. of SPIE, Vol 298, Real Time Signal Processing*, pages 241–248, 1981.

[Smi 85] B. Smith. The Architecture of HEP. In J. S. Kowalik, editor, *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, pages 41–55. The MIT Press, 1985.

[Smi 90] B. Smith. The End of Architecture: Keynote Address presented at the 17th Annual Symposium on Computer Architecture. *ACM SIGARCH, Computer Architecture News*, 18(4):10–17, December 1990.

[SmL 84] A. Smith and J. Lee. Branch Prediction Strategies and Branch Target Buffer Design. *Computer*, 17(11):6–22, Jan. 1984.

[Sri 86] V. P. Srini. An Architectural Comparision of Dataflow Systems. *Computer*, 19(3):68–88, 1986.

[Sta 86a] C. A. Staley. Design and Analysis of the CCMP: A Highly Expandable Shared Memory Parallel Computer . Technical report, Department of Electrical and Computer Engineering, Santa Barbara, CA, August 1986. Ph.D Dissertation.

[Sta 86b] C. A. Staley and S. F. Butner. A feasibility study and simulation of the circulating context multiprocessor. In *Proc. of Intl. Conf. on Parallel Processing*, pages 455–462, 1986.

[Ste 89] G. B. Steven, S. M. Gray, and R. G. Adams. HARP: A Parallel Pipelined RISC Processor. *Microprocessors and Microsystems*, pages 579–587, 1989.

[Sti 91b] T. Stiemerling. Design and Simulation of an MIMD Shared Memory Multiprocessor with Interleaved Instruction Streams. Technical Report CST-85-91, Edinburgh University, Edinburgh, UK, 1991. Ph.D Dissertation.

[Sto 90] H. S. Stone. *High Performance Computer Architecture - Second Edition.* Addison-Wesley Publishing Company, 1990.

[Tho 64] J. E. Thorton. Parallel Operation in the Control Data 6600. In *Proc. Fall Joint Computer Conf. 26*, pages 33–40, 1964.

[Top 88] N. P. Topham et al. Context Flow: An Alternative to Conventional Pipelined Architectures. *Journal of Supercomputing*, 2(1):29–53, 1988.

[Tra 91] K. R. Traub. Monsoon: Dataflow Architectures Demystified. In *Proceedings of the 13th IMACS World Congress on Computation and Applied Mathematics*, 1991.

[Tre 91] P. C. Treleaven. Data Driven and Demand Driven Computer Architecture. *ACM Computing Surveys*, 14(1):93–143, 1982.

[Veen 86] A. H. Veen. Dataflow Machine Architecture. *ACM Computing Surveys*, 18(4):365–396, 1986.

[Veg 84] S. R. Vegdahl. A Survey of Proposed Architectures for the Execution of Functional Languages. *IEEE Trans. on Computers*, C-33(12):1050–1071, 1984.

[Vin 90] H. M. Vin and F. Berman. Architectural Support for the Efficient Data-Driven Evaluation Scheme. In *2nd Annual ACM Symp. on Parallel Algorithms and Architectures*, pages 179–187, 1990.

[Vogt 88] C. Vogt and K. von der Heide. Die Puffermaschine: Entwurf und Realisierung einer fehlertoleranten, objektorientierten Rechnerarchitektur, Teil 1: Grundlegende Konzepte. Technical Report 387, FFM, Forschungsinstitut fuer Funk und Mathematik, D-5307 Wachtberg-Werthhoven, West Germany, Dec. 1988. (in German).

[Vogt 90] C. Vogt. A Buffer-Based Method for Storage Allocation in an Object-Oriented System. *IEEE Trans. on Computers*, C-39(3):375–383, 1990.

[VdH 86] K. von der Heide. A General Purpose Pipelined Ring Architecture. *Lecture Notes on Computer Science*, 237:198–205, 1986.

[Wang 91] L. T. Wang and C. Wu. Distributed Instruction Set Computer Architexture. *IEEE Transactions on Computers*, C-40(8):915–934, August 1991.

[Web 89] W. D. Weber and A. Gupta. Exploiting the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 273–280. IEEE, 1989.

[Wei 88] Y. H. Wei and J. L. Gaudiot. Demand-Driven Interpretation of FP Programs on a Data Flow Multiprocessor. *IEEE Trans. on Computers*, C-37(8):946–966, 1988.

[Wu 90] H. D. Wu and L. R. Thorelli. Extending Dataflow Principles for Multiprocessing. In *Proc. of Intl. Conf. on Parallel Processing*, pages II53–60, 1990.

I declare that this thesis – Latency-Directed Multithreaded Computation and Its Architectural Support – was written by myself, and that the work described herein is my own.

Hamburg, 23. August 1993 Xiaoming Fan

Curriculum Vitae

March 29, 1961	born in Hangzhou/Zhejiang, China
Sept. 1967 - Jan. 1973	Primary School in Hangzhou, China
Feb. 1973 - Juli 1978	Second and High School in Hangzhou, China
Sept. 1978 - Juli 1982	Undergraduate study in Computer Science, the University of Science and Technology of China, Hefei, Anhui, China (B.S. Degree)
Sept. 1982 - Dec. 1984	Graduate study in Computer Science, the University of Science and Technology of China, Hefei, Anhui, China (M.S. Degree)
Jan. 1985 - April 1988	Assistant at the Dept. of Computer Science, the University of Science and Technology of China, Hefei, Anhui, China
April 1988 - March 1989	Visiting scholar at the Dept. of Computer Science, the University of Hamburg
April 1989 - March 1993	Research assistant at the Dept. of Computer Science, the University of Hamburg
April 1993 -	Employee in Delta T, GmbH, Hamburg