1

Microl

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

STUDY OF PIPELINE PROCESSING SYSTEMS

by

Anirban Basu, M.Tech.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of

Master of Computer Science

School of Computer Science

Carleton University
Ottawa, Ontario
March 16, 1987

© copyright
1987, Anirban Basu

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication: ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-39413-7

6

The undersigned recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis

Study of Pipeline Processing Systems

submitted by Anirban Basu, M.Tech.

in partial fulfilment of the requirements for
the degree of **Master of Computer Science**

M. R. Pugh

Thesis Supervisor

J. R. Pugh

Chairman, School of Computer Science

Carleton University

May 14 1987

Date

Abstract

The concern of this thesis is with problems occurring in pipeline processing systems. Both Instruction Pipeline and Arithmetic Pipeline are investigated and solutions are presented for some of the fundamental problems occurring in these systems.

Solutions to the branch problem in an Instruction Pipeline are usually based on predicting whether or not a branch will be taken. As the performance of an Instruction Pipeline depends on the prediction accuracy, an expression is deduced in this thesis, relating it to the efficiency. To the author's knowledge, this is the first time that such an expression is given.

For overcoming the problem of inflexibility of conventional Arithmetic Pipelines, an Adaptable Pipeline is proposed whose design is based on VLSI processing elements. The proposed design overcomes the drawbacks of all the other designs and is implementable with commercially available Transputers.

In a Pipeline Processor, fault in any stage causes disruption of flow of instruction/data through the pipeline. Although the issue is crucial, this is the first time that a design of a Fault Tolerant Pipeline Processor is described whose operation is unimpaired even when a stage fails.

Finally, a nomenclature is proposed for describing the salient architectural features of all the different types of parallel processing systems that have evolved.

Acknowledgements

I would take this opportunity to express my gratitude to my thesis supervisor Prof. Nicola Santoro for his valuable suggestions and constant encouragement during the course of this research work. In spite of his various preoccupations, he devoted time for stimulating discussions, whenever the need arose.

I would like to thank the Canadian Commonwealth Scholarship and Fellowship Committee for funding my studies at Carleton University. Without their financial assistance I could not have pursued the work reported in this thesis. To the Ministry of Education, Government of India, I remain grateful for nominating me for the Canadian Commonwealth Scholarship.

I should also thank my fellow students in the School of Computer Science for coming forward whenever I needed their assistance.

Table of Contents

Acceptance Sheet.....	ii
Abstract.....	iii
Acknowledgements.....	iv
Table of Contents.....	v
List of Tables.....	vii
List of Figures.....	viii
1. Introduction.....	1
2. Instruction Pipeline.....	10
2.1 Approaches to the Branch Problem.....	14
2.1.1 Loop Buffers.....	14
2.1.2 Multiple Instruction Streams.....	15
2.1.3 Prefetch Branch Target.....	16
2.1.4 Delayed Branch.....	16
2.1.5 Taken/Not Taken Switch.....	19
2.1.6 Branch Target Buffer.....	19
2.2 Branch Prediction Strategies.....	22
2.2.1 Static Prediction Strategies.....	23
2.2.2 Dynamic Prediction Strategies.....	24
3. Impact of Prediction Accuracy on the Performance of an Instruction Pipeline.....	26
3.1 Efficiency with no Precedence Constraint.....	29
3.2 Efficiency with Procedural Dependency.....	31
3.3 Effect of Prediction Accuracy.....	35

4. Arithmetic Pipeline.....	38
4.1 Collision Prevention and Reservation Table.....	39
4.2 Types of Arithmetic Pipeline.....	42
4.2.1 Static Multifunctional Pipeline.....	42
4.2.2 Dynamic Multifunctional Pipeline.....	47
4.3 Vector Processors.....	49
5. Design of an Adaptable Pipeline based on Transputers.....	52
5.1 Hardware Organisation of the Adaptable Pipeline.....	53
5.2 Format of an Instruction.....	58
5.3 Operation of the Adaptable Pipeline.....	62
5.4 Organisation of a Pipeline Stage.....	64
5.4.1 Organisation of the First Stage.....	68
6. A Fault Tolerant Pipeline Processor.....	71
6.1 Hardware Organisation of the Fault Tolerant Pipeline Processor.....	73
6.2 Format of an Instruction.....	80
6.3 Operation of the Fault Tolerant Pipeline Processor.....	82
6.4 Organisation of a Pipeline Stage.....	85
7. A Nomenclature for Parallel Processing Systems.....	89
7.1 Inadequacies of Existing Taxonomies.....	90
7.2 Design Decisions and the Taxonomical Tree.....	92
7.3 System Architectures and the Proposed Taxonomy.....	97
8. Concluding Remarks.....	102
References.....	106

List of Tables

Table	Description	Page
4.1	Inputs required for the four functions.....	46

List of Figures

Figure	Description	Page
2.1	Normal and Delayed Jump.....	18
2.2	Organisation of a Branch Target Buffer.....	20
3.1	Space-time diagram when processing L instructions in a four segment pipeline.....	28
3.2	Variation of <i>Efficiency</i> with <i>Prediction Accuracy</i> for a Pipeline with equal segment times.....	37
4.1	A Pipeline and the Reservation Table for a computational process.....	40
4.2	The cells and its logic equations.....	44
4.3	A four function Arithmetic Pipeline.....	45
4.4	Four functional configurations in the TI-ASC Arithmetic Pipeline.....	48
5.1	Links of a Transputer.....	55
5.2	Hardware Organisation of an Adaptable Pipeline.....	55
5.3	Instruction Format.....	58
5.4	Data Flow Graph of a computational process and its execution in the Adaptable Pipeline.....	60
5.5	Channel Organisation.....	65
5.6	Flow of operations in a Pipeline Stage.....	67
5.7	Flow of operations in the First Stage.....	70
6.1	Links of a Transputer.....	73

6.2	A Fault Tolerant Pipeline Processor.....	75
6.3	Logic inside the Transfer <u>Blocks</u>	78
6.4	Instruction Format.....	80
6.5	Channel Organisation.....	86
6.6	Flow of operations in a Pipeline Stage.....	87
7.1	The Taxonomical Tree.....	93

Chapter One

Introduction

Two design philosophies have been followed in the design of parallel processing systems to reduce the execution time of a program: *parallelism* and *pipelining*. Parallelism reduces program execution time by replicating a hardware structure more than once and by having all structures work simultaneously on different parts of the program. Pipelining refers to the segmentation of a repeated computational process into several subprocesses each of which is executed on a dedicated autonomous unit (called stage or segment) operating concurrently with the others; successive processes can be carried out in an overlapped mode analogous to an industrial assembly line.

Application of pipelining to the design of a computer system increases performance. If a function which takes C ns. when executed in a non-pipelined straightforward computer system is partitioned into subfunctions for repeated execution in N stages of a pipeline processor, then the output can be obtained every C/N ns., giving an N fold increase in performance. Needless to say, there are problems which tend to limit this maximum achievable rate.

In a high speed computer system, the technique of pipelining may be applied at two levels: in the design of the instruction processing unit, in which the different steps of the instruction cycle are performed in overlapped time spans, and in the implementation of complex arithmetic operations which are broken into simpler operations for execution in different pipeline segments. A system which utilises the first level of pipelining is referred to in the literature as Instruction Pipeline while

Arithmetic Pipeline is used in the second case.

In an Instruction Pipeline, the instruction processing function is segmented into subfunctions like instruction fetch, instruction decode, operand fetch and execute which are carried out by dedicated pipeline stages. An Instruction Pipeline achieves maximum throughput when processing independent instructions. Problems arise when some instructions require the results of previously issued instructions before they can be executed. Delays occur while the results of these previously issued instructions are determined. Branch instructions have the most adverse effect on the performance of an Instruction Pipeline. Although many methods have been proposed to overcome the effect of branch instructions, its adverse affect cannot be nullified completely. Most of the recent methods of solution are based on predicting whether or not a branch will be taken. If the prediction is correct, computation continues without delay. The performance of an Instruction Pipeline therefore depends on the accuracy of prediction.

In an Arithmetic Pipeline, a complex arithmetic operation is broken into simpler operations for execution in different segments of the pipeline. The technique of pipelining has found widespread application in the realisation of arithmetic algorithms for computing floating point multiplication, division, square root etc.. However, the major problem of Arithmetic Pipeline is that it tends to be dedicated to the particular type of computation for which it is designed. Any disparity between the pipeline and the algorithm of a computational process introduces a large time overhead thereby limiting their applicability. To broaden the range of their cost-effective applications, Arithmetic Pipelines employ software or firmware controllable reconfigurations of the hardware resources. The general idea is to reconfigure the resources to reduce the dissimilarity between the Arithmetic Pipeline and the sequences of operations required by the different arithmetic computations. In existing pipeline systems, reconfiguration is used in a limited

way, by bypassing or chaining pipeline stages. A pipeline system which permits interconnections among pipeline stages can serve a variety of functions, but such multifunctional Arithmetic Pipelines require elaborate control mechanisms. Most existing multifunctional pipelines such as CRAY-1 [12], MU 5 [27][28], TI ASC [62] are static i.e., at any time instant only one configuration is active. To perform transition from one sequence of operations to another, the pipeline has to be reconfigured. Since during reconfiguration no processing is performed, the reconfiguration time is a pure time overhead which should be minimal. Obviously, throughput can be much higher in pipeline systems which permit overlapped processing among several active configurations simultaneously. Besides improvement in throughput, the applicability of a pipeline system can be expanded if it is provided with the following features:

- each pipeline stage is capable of executing any operation.
- the system forms pipelines with a variable number of stages. This means that each computational process activates the number of stages in the pipeline which matches the number of consecutive operations it implements i.e., an instruction implementing w operations will pass through w stages and complete its execution.
- the system provides for fast exchanges of temporary results between pipeline stages.
- the system adapts to a varying number of parallel program streams by partitioning resources to form a variable number of parallel pipelines. For example, a system with ten stages may reconfigure to form one pipeline with six stages and two pipelines with two stages each depending on the number of parallel program streams.

A system with the above architectural features is called an *Adaptable Pipeline*. As discussed by Alexandridis in [2], a pipeline computer system with such characteristics can provide very low response time and is capable of fulfilling the requirements of mission critical military applications.

In addition to performance, an important issue in the design of any computer system is its ability to continue operation in spite of failure of some active elements of the system. In a pipelined system, failure of a stage may be catastrophic as it may cause disruption of flow of instruction or data through the entire pipeline. For achieving fault tolerance, a pipeline computer system should have the ability to bypass the faulty stage so that computation can continue.

This thesis is concerned with problems occurring in pipeline processing systems. Both, Instruction Pipeline and Arithmetic Pipeline are investigated and solutions are presented for some of the fundamental problems occurring in these systems.

The major problem in Instruction Pipeline is caused by the presence of conditional branch instructions in an instruction stream. Performance degradation due to branches can be reduced in a number of ways namely loop buffers, multiple instruction streams, prefetch branch target, prepare to branch, delayed branch, taken/not taken switch and the branch target buffer. A number of solutions to the branch problem attempt to predict whether or not a branch will be taken. If the prediction is correct, then the execution of the correct sequence of instructions continues without delay. If however, the prediction is wrong, the initiated instructions have to be 'squashed', that is these instructions have to be disabled so that they do not change the program state. The performance of an Instruction Pipeline depends on the *prediction accuracy* which is defined as the probability of correct branch prediction. The sole aim of all the prediction strategies that have been investigated [47][57] is to obtain a value of 1 for the prediction accuracy. However, the results of experiments [47] show that this goal is not achievable. A question then arises. How sensitive is the performance of an Instruction Pipeline to the prediction accuracy? For answering this question, the performance of an Instruction

Pipeline is analysed in this thesis, and a quantitative measure of performance given relating it to the probability of correct branch prediction. Although the performance of pipeline processors has been studied by a number of researchers [6][8][24][51], this is the first time that an expression is provided relating efficiency of an Instruction Pipeline and the prediction accuracy. Here the expression for *efficiency* or *utilisation* of pipeline segments is deduced from the space-time or geometric model [8] for a pipeline with unequal segment times incorporating the effect of conditional branches and the probability of correct branch prediction. Although, it is extremely difficult to obtain a value of 1 for the prediction accuracy, this study indicates that a prediction accuracy of 0.8 can give reasonably good values of utilisation of pipeline segments.

As stated before, conventional Arithmetic Pipelines suffer from the problem of inflexibility. An Adaptable Pipeline can overcome this drawback and can provide very high throughput. In this thesis, the design of an Adaptable Pipeline is proposed which is based on commercially available VLSI processor chips called the Transputers [63]. The concept of adaptable pipelining was introduced in an earlier architecture proposed by Kartashev and Kartashev in 1979 and described in [35][36]. Their design was based on special purpose LSI chips called Dynamic Computer Groups (DC Groups) [34]. Despite the similarity in concept, the system proposed by Kartashev and Kartashev is in many ways a pre-VLSI architecture. Widespread use had been made of switches, resulting in a complex control system. Also, the pipeline designed by them can be used only when the computational process does not require internal looping i.e., there is no necessity for the results to be fed back as inputs. The design proposed here overcomes these drawbacks.

The main considerations which led to the choice of Transputers as the basic building block in this design were that these VLSI components are commercially

available, it supports concurrent message passing and have architectural support for OCCAM [31]. The latter feature enables efficient execution of reconfiguration software [38], if written in this concurrent programming language.

Besides making provision for internal looping, the proposed design based on Transputers puts greater emphasis on limiting the complexity of the interconnection switches and on increasing the versatility of the pipeline stages thereby making the Adaptable Pipeline useful in a wide range of processing applications.

Such a pipeline can execute the following computational processes simultaneously:

- a complex arithmetic operation like floating point addition, floating point multiplication
- a vector instruction
- a task, which is decomposed into subtasks each of which consists of a sequence of instructions. The same data stream is acted on by a cascade of subtasks, one after another in a pipelined mode. The data stream passes through the first stage of the pipeline and the results are sent to the second stage. The second stage then passes the refined results to the third stage and so on. This has been referred to as processor pipelining in [26].

Further, the Adaptable Pipeline is capable of switching its resources into different states, where each state is specified by the number of concurrent pipelines and the number of stages in each pipeline. For example in one state, the entire resource may go into the formation of three pipelines, each containing five stages. In another state, the system may form two pipelines with four stages each and two pipelines with three stages each. Such dynamic redistribution of resources facilitates the concurrent execution of instruction streams and also permits the organisation of vector chaining [52].

Because of their important role in mission critical applications it is

necessary that a pipeline processor be fault tolerant. For achieving fault tolerance, a pipeline processor should have the ability to bypass the faulty stage so that computation can continue. Although the problem is very important, it is unfortunate that very little effort has been invested in this direction. In this thesis, the design of a *Fault Tolerant Pipeline Processor* is described whose operation is unimpaired even when a pipeline segment fails. While, research has been conducted into the design of fault tolerant multiprocessor systems [1][43][56], to the author's knowledge this is the first time that a design of a fault tolerant pipeline processor is proposed. Like the Adaptable Pipeline, the design of the Fault Tolerant Pipeline Processor is based on Transputers but has additional hardware for incorporating the fault tolerance capability.

Besides Instruction and Arithmetic Pipeline, some parallel processing systems have been proposed which employ the technique of pipelining to reduce the execution time of an algorithm but cannot be categorised as a conventional Instruction or Arithmetic Pipeline. Systolic Arrays and Data Flow computers are examples. In a Systolic Array, data flows through an array of processing elements in a pipelined way. Data Flow computers basically apply the technique of pipelining with the only difference that the order of instruction execution is not implied by the order of instructions in storage. Although the primary aim of this thesis is to investigate issues involved in the design and performance of pipeline processing systems, in the last part of the thesis the characteristics of all other types of parallel processing systems are described and a nomenclature is proposed to describe all the high speed computing systems that have evolved.

The parallel processing systems that have been developed are referred to in the literature [22] by terms such as Systolic Arrays, Array Processor, Associative Processor, Pipeline Processor, Multiprocessor, Multifunction Processor, Data Flow

Processor, Demand Driven Processor. Unfortunately however, these terms do not give precise information about the dominant characteristic features or the capabilities of these systems. For example, a computer system consisting of a number of CPUs is termed a multiprocessor or a multicomputer system even though it may have any one of the strongly contrasting control patterns as centralised control, distributed control, data flow control. Moreover, although a few of the most recently proposed designs fall precisely into the aforementioned types, most current designs are hybrids which combine the concepts from two or more of these. For example, Cray X-MP [7] is a multiprocessor system composed of two tightly coupled CPUs where each CPU is itself a pipeline processor. Today because of the low cost of hardware, such hybridation has become a rule rather than exception and there is no way in which the dominant characteristics of such hybrid systems can be described by the use of simple terms.

Noting the many parallel processing systems that have been designed, it is essential that a taxonomy be developed which would bring out the diversity in their designs as well as enable the specification in simple terms the salient features of these high speed systems. The major problem in formulating such a taxonomy is choosing the minimum amount of detail that should be incorporated in the classification scheme so as to make the nomenclature complete as well as compact. A taxonomy which meets these two conflicting requirements will permit very compressed descriptions of the systems. It will employ only those features of the architecture that are of interest, ignoring the rest. There have been some attempts to develop such a taxonomy [5][15][16][18][21][41] but all these have been found to be inadequate. Their drawbacks are discussed and a new classification of parallel processing systems is proposed with a view to developing a nomenclature based on system characteristics which are germane to these architectures.

The thesis is organised as follows.

In Chapter 2 Instruction Pipeline is considered. The basic concepts are briefly introduced and the branch problem together with the proposed solution methods are discussed. Most of the recent methods of solution are based on predicting whether or not a branch will be taken and a number of prediction strategies have been proposed. Some of these are discussed.

In Chapter 3, an Instruction Pipeline is analysed and a methodology for studying the impact of prediction accuracy on its performance is given.

In Chapter 4, the concepts of arithmetic pipelining is discussed and the problems of reconfiguration of a conventional Arithmetic Pipeline are investigated.

In Chapter 5, the design of an Adaptable Pipeline is presented. The design is based on commercially available VLSI processing elements called Transputers. The mode of operation of the proposed system is illustrated when it is applied to the processing of complex arithmetic functions.

In Chapter 6, the design of a Fault Tolerant Pipeline Processor is discussed whose operation is not affected by the complete failure of a pipeline segment.

In Chapter 7, the characteristics of different parallel processing systems are described and a nomenclature for these systems is presented which is based on their dominant architectural features.

Finally, Chapter 8 concludes the thesis and gives directions for further research.

Chapter Two

Instruction Pipeline

In an Instruction Pipeline, the instruction processing functions are segmented into several subfunctions such as instruction fetch, instruction decode, operand fetch and execute which are carried out by dedicated autonomous units, called pipeline segments. Amdahl 470V/6 [3], IBM 360/91 [4], MU5 [27][28] are examples of systems employing instruction pipelining. For understanding the advantage of instruction pipelining, consider an Instruction Pipeline composed of four stages: instruction fetch, instruction decode, operand fetch and execute. Then, while the execute stage is processing the first instruction, the operand fetch stage fetches the operands needed by the second instruction, the instruction decode stage prepares the different operations for the third instruction and the instruction fetch module fetches the fourth instruction i.e., four instructions are processed simultaneously in the four stages of the Instruction Pipeline. If successive instructions are independent, then the rate of processing instructions is limited only by the speed of the slowest segment of the pipeline referred to as the 'bottleneck' but the time to complete a particular instruction is given by the sum of the times in all the stages.

The number of segments in an Instruction Pipeline varies between machines depending on the amount of processing required to be performed at each stage. If a simple operation is performed in each stage, then operation in that stage can be performed quickly and throughput rate is high. As each stage is made simpler, a larger number of stages are required. Hardware cost and the number of instructions present in the pipeline at any one time increases in this case. This can decrease utilisation of

pipeline stages when successive instructions are not independent. i.e., when some instructions require the results of previously issued instructions before they can be executed. Increasing the number of stages also creates problems in interrupt handling. Choosing the number of stages from the viewpoint of performance/cost ratio has been discussed in [46].

The throughput of an Instruction Pipeline is determined by the speed of the bottleneck segment. As discussed in [52], the speed of the bottleneck stage can be improved by subdivision of the bottleneck element or by putting facilities in parallel. In an Instruction Pipeline, the bottleneck is in memory access stage and in the execute stage. The simultaneous accessing of instructions and operands from memory may create memory reference conflict and a fetch may have to be delayed for a complete memory cycle. Unless memory access conflicts are reduced, the memory access stage may become a bottleneck of such a large magnitude that the throughput of the processor is not much improved by pipelining. Whenever the execution time of a stage is not fixed, buffering becomes essential for smoothing out the flow through the pipeline. Besides interleaving memory addresses, memory access related stages in an Instruction Pipeline are provided with buffers for storing instructions and operands to prevent any disruption of flow through the pipeline. CDC Star 100 [10], IBM 360/91 [4], TI ASC [62] all have instruction and operand buffers. Amdahl 470V/6 [3] uses a high speed 16 byte cache to reduce memory reference. Ramamoorthy and Wah [53] have suggested an organisation of an interleaved memory system which utilise a finite buffer space for storage of memory requests. They have proposed a design of a scheduling algorithm which allows a finite set of memory requests to be processed in the minimum expected completion time. In their study, CDC 7600, a pipelined processor have been used for generation of non-random memory requests for the trace driven simulation. They have also studied the

effect of separation of instruction and data areas to reduce memory access time. But for the specific class of problems chosen for the study and for CDC 7600, the separation showed a minimal improvement. However, it is felt that this aspect can be investigated further.

Instruction Pipeline achieves a high performance/cost ratio when processing independent instructions. Problems arise in an Instruction Pipeline, when some instructions require the results of previous instructions before they can be executed and delays will occur while the results of these instructions are determined. This aspect has been discussed further in this chapter. Besides the disruption of flow caused by precedence constraints among the successive instructions, the performance of an Instruction Pipeline suffers due to occurrence of interrupts. If an interrupt occurs during execution of instruction i , the interrupt has to be serviced before processing instruction $i+1$. However, when an interrupt occurs there may be several instructions in various stages of processing. Execution of these instructions has to wait till servicing of the interrupt is completed. Recovering these instructions after servicing the interrupt imposes a complex problem.

As already stated, the performance of an Instruction Pipeline suffers due to precedence constraints in the instruction stream. There may be three kinds of dependencies among instructions in a program: operational dependency, data dependency and procedural dependency.

An operational dependency arises when an arithmetic logic unit required for executing the operation specified in an instruction is busy. Its treatment involves providing multiple execution units which can execute the arithmetic/logic operations as required in successive instructions. If such hardware investments are too costly, then use can be made of virtual functional units (or Reservation Stations in IBM terminology)

with the tagged register scheme suggested by Tomasulo [60]. In this scheme, instead of not issuing an instruction to a busy execution unit, the request is put in a FIFO queue attached to the execution unit. When the latter becomes free, it selects the first instruction in its queue (if any). This is the idea followed in IBM 360/91.

A data dependency arises in a sequence of instructions when an instruction has a sink operand which is the source operand of any successive instruction. If $i+1$ -th instruction is data dependent on the i -th instruction then the $i+1$ -th instruction can not fetch operand from memory until the i -th instruction writes the result. One way of avoiding the delay due to data dependence is by using a control structure which allows independent instructions after the conflicting one to continue. This is referred to as a fully asynchronous control structure [52]. It allows out of order termination of instructions but controls the execution of instructions so that the results obtained are the same as that which would have resulted from strict sequential execution. In an Instruction Pipeline, lookahead is easy to implement because several successive instructions are in the instruction buffer and in some state of preparation for execution. For overcoming the problem caused by data dependency, an Instruction Pipeline may follow a centralised approach as in SCOREBOARD of CDC 6600 [58], where instructions are initiated after checking the information about operand availability and also the availability of an arithmetic/logic unit. Alternatively, a decentralised policy such as that followed in the Common Data Bus (CDB) in IBM 360/91 may be applied. From this discussion it can be concluded that there are methods which can fully overcome the problem created by data dependency among successive instructions.

Of the three types of dependencies in the instruction stream, procedural dependence is the most serious, as its adverse effect on the performance of an Instruction Pipeline cannot be fully overcome. A procedural dependency is a dependency in the

specification of the instruction sequence. If it is assumed that instructions are arranged 'in line' sequence i.e., one instruction in the sequence followed by the next, then only a branch out of sequence causes a dependency. Thus given a sequence of instructions (ascending in time) in an instruction stream, if I_i is a branch instruction, then for $\{ I_1, \dots, I_i, \dots, I_m \}$ all instructions $I_j, i < j \leq m$ are dependent on I_i . If I_i is an unconditional branch instruction, then the dependency can be quickly resolved and the problem overcome by refetching the correct sequence. On the other hand, if I_i is a conditional branch instruction the correct sequence of instructions can not be initiated till the data test is made and branch path determined. Only then can the dependency be resolved. In this case, the degradation is serious and it causes 'bubbles' in the pipeline, as some stages become idle. Various techniques have been proposed which attempt to prevent the disruption of flow of instructions through the pipeline due to the presence of branch instructions in an instruction stream. These are now discussed.

2.1 Approaches to the Branch Problem

The salient features of the methods that have been proposed to overcome the branch problem in an Instruction Pipeline are described under the following headings.

2.1.1 Loop Buffers

A loop buffer is a small, very high speed buffer maintained by the instruction fetch stage of the Instruction Pipeline, and contains a set of sequential

instructions. Multiple loop buffers contain n sequences, one per buffer, but the contents of the buffers may not be contiguous. The loop buffer serves two purposes. First it contains instructions sequentially ahead of the current instruction fetch address thereby ensuring the availability of instructions without memory access. Second, it can recognise when the target of a branch (including backward branches) falls within its contents and deliver the appropriate sequence of instructions without accessing memory. All instructions in a loop can be fetched from this buffer; hence the name 'loop buffer'.

Among the machines using a loop buffer are the CDC Star-100 with a buffer of 256 bytes [10], the CDC 6600 with 60 bytes [58] and CDC-7600 with 12, 60 bit words [11].

2.1.2 Multiple Instruction Streams

This scheme follows a brute force approach and replicates the initial stages of the pipeline so that both the sequential instruction and the potential branch target can be fetched, decoded and processed. However, this approach gives rise to three problems. The first is that the branch target cannot be fetched until its address is determined, which may require a computation such as adding a displacement to both a base and index register. This computation requires time even when operands are available. Delays may occur when operands are not available, such as when an operand is the result of an uncompleted instruction or when a memory fetch is required. Replicating the initial stages of the pipeline gives rise to resource contention. For example, a register file may be accessed from more than stage. Also there is an increase in memory traffic giving rise to memory contention.

The second problem in replicating the initial stages of the pipeline is that, if

instruction i is a branch instruction then additional branch instructions may enter either of the two pipelines before i can be resolved as taken or not taken and its target determined. Riseman and Foster [54] found that for a typical pipeline, more than two branches would have to be processed this way to yield a significant improvement. The hardware required for the purpose would therefore be impractical.

The third drawback of this method is that the cost of replicating initial stages of the pipeline may be substantial making this mechanism of questionable value.

2.1.3 Prefetch Branch Target

Rather than replicate several initial stages of the pipeline, this method is based on duplicating enough logic to prefetch the branch target. That is, when a branch instruction is encountered, a special mechanism calculates and prefetches the target of the branch. If the branch is found to be taken, the target instruction can be immediately loaded into the instruction decode stage of the pipeline thereby avoiding additional delay for instruction fetch. Several such prefetches can be accumulated along the main instruction sequence, but since the secondary (prefetched) sequences are not decoded, no additional prefetches can be generated there.

The IBM 360/91 uses this mechanism to prefetch a double word target.

2.1.4 Delayed Branch

The problem with a branch is that if instruction i is a branch which is taken, then instruction $i+1$ will be out of sequence. The instruction set of an Instruction Pipeline can be specified such that a branch is defined to affect the address not of

instruction $i+1$ but of instruction $i+k$, where $k > 1$. For example, consider a sequence of instructions $I_1, \dots, I_{10}$. Let I_4 be conditional branch instruction whose target is I_9 . Assume that branches are delayed two instruction times, making $k = 2$ and that I_4 is a taken branch. Then the actual sequence of instruction initiation would be $I_1, I_2, I_3, I_4, I_5, I_6, I_9, I_{10}$.

If k , the branch delay, is equal to or larger than the number of stages preceding the stage in which the branch is executed, then the instruction fetch stage will always be provided with the correct address from which to fetch the next instruction except when interrupts occur.

Several problems are encountered, when designing an Instruction Pipeline using delayed branch technique. The most important of these is that the code incorporating delayed effects in branch instructions have to be compiler-generated, with the consequent need for a very efficient compiler. The optimisation phase of the compiler has to rearrange the sequence of instructions to perform the delayed branch.

The following example illustrates the method. Figure 2.1(a) shows a sequence of instructions which in machines with normal jump would be executed in the order 100, 101, 102, 105. If the code is executed on a pipeline using a delayed jump technique then the compiler includes an optimiser that rearranges the sequence of instructions which is 100, 101, 102, 105 (shown in Figure 2.1(b)). Because the instruction following the jump is always executed and the Jump at 101 is not dependent on the ADD at 102, this sequence is equivalent to the original program segment of Figure 2.1(a).

However, it should be noted that achieving a value of $k > 2$ is extremely difficult. Therefore the method will work in a pipeline with a very small number of

Address	(a) Normal	Jump	(b) Delayed	Jump
100	LOAD	X,A	LOAD	X,A
101	ADD	1,A	JUMP	105
102	JUMP	105	ADD	1,A
103	ADD	A,B	ADD	A,B
104	SUB	C,B	SUB	C,B
105	STORE	A,Z	STORE	A,Z

Figure 2.1 Normal and Delayed Jump

stages but fails in an Instruction Pipeline of moderate length. The Delayed Branch has been proposed for RISC computer [49].

2.1.5 Taken/Not Taken Switch

The method is based on predicting whether or not a branch will be taken. A mechanism which specifies whether a branch is or is not likely to be taken is called the taken/not taken switch. The idea is that one or more bits are associated with every instruction in the cache memory. The setting of these bits determines whether the branch is predicted to be taken or not. After the branch is resolved, the values of the bits may be reset in the cache to reflect the prediction for the next time.

In the taken/not taken switch proposed for the S-1 computer [64], two bits are stored with each instruction. One bit specifies whether the jump should be predicted (the Jump bit) and the other tells whether the last prediction was wrong (the Wrong bit). Two wrong predictions in a row cause the jump bit to be changed. This mechanism still encounters delays due to target address computation and out of sequence fetch.

2.1.6 Branch Target Buffer

The Branch Target Buffer (BTB) is a small cache memory associated with the instruction fetch stage of the pipeline. The buffer is used to predict whether the branch will be taken and if so what its target will be. The instruction fetch stage then continues by fetching instructions from the predicted address.

The BTB (Figure 2.2) retains three tuples, which contains the address of a previously executed instruction, information that permits a prediction as to whether or

Figure 2.2 Organisation of a Branch Target Buffer

not the branch instruction will be taken and the most recent target address for that branch. The BTB functions as follows: the instruction fetch stage compares the instruction address against the instruction addresses in the BTB. If there is a match, then a prediction is made as to whether the branch is likely to be taken. If the prediction is that the branch will actually occur, then the target address field is used to select the next instruction fetch address. When the branch is actually resolved, at the execute stage, the BTB can be updated with the corrected prediction information and target address. Since the BTB can be used for every instruction fetch, it can have as many predictions as there are uncompleted instructions in the pipeline.

The major problem in the design of the BTB is the selection of the algorithm that predicts whether or not the branch will be taken. The size of the BTB is another issue. Holgate and Ibbett [24] have studied the BTB design effectiveness for the MU5 computer system, which actually implements a Branch Target Buffer, but calls it 'Jump Trace' mechanism. The 'Jump Trace' mechanism requires a small associative memory which remembers the results of upto eight previously executed control transfers. Thus when a particular control transfer is executed for the first time, the address of the control transfer is loaded into a store location in the associative 'jumped from address' field of the Jump Trace and the target address of the control transfer is loaded into the corresponding location in the 'jumped to address' field. Instructions are normally fetched from sequential addresses, but every address generated by Instruction Counter are checked by association, against the 'jumped from address' and if a match is found the corresponding 'jumped to address' is read out, so that prefetching of the new sequence takes place instead. This system prefetches the correct sequence of instructions in a loop, on all but the first and the last times through, thus enabling processing to continue in most cases.

The many aspects involved in the design of the Branch Target Buffer has been discussed in [47].

Among the methods discussed here, the Multiple Instruction Stream method is capable of eliminating any disruption of flow through the pipeline due to branch instructions hundred percent of the time. But as discussed in section 2.1.2, this method has a number of problems besides requiring an impractically large amount of hardware. Although, hardware cost is not a major hindrance these days, increase in hardware involves increase in control complexity making the method difficult to implement. According to Lee and Smith in [47], the Branch Target Buffer based on branch prediction is the most acceptable solution to the branch problem. The advantage of this method has been supported by studies made by McFarling and Hennessy [48].

From the above discussion it is clear, that an acceptable cost effective solution to the branch problem may be based on predicting the branch direction and initiating instructions so that execution can proceed without penalty if the prediction is correct. However, if the prediction is wrong, the initiated instructions must be 'squashed' i.e., these instructions in the pipeline have to be disabled so that they do not change the program state. Obviously, the performance of the Instruction Pipeline depends on the accuracy of branch prediction. A number of strategies have been proposed in [47] and [57], for predicting the direction of a branch. Some of these are now discussed.

2.2 Branch Prediction Strategies

A number of strategies have been proposed with the goal of maximising

the likelihood of correctly predicting the outcome of a branch. Strategies are divided into two basic categories, depending on whether or not history is used to make a prediction. For example, a straightforward method for branch prediction is to predict that branches are either always taken or not taken. Since such a strategy always makes the same prediction every time a branch instruction is encountered, it is called Static. On the other hand, it has been observed [47] that the likelihood of a conditional branch instruction at a particular location being taken is highly dependent on the way the same branch was decided previously. This leads to Dynamic prediction strategies in which the prediction varies based on branch history. Both Static and Dynamic prediction strategies are now discussed.

2.2.1 Static Prediction Strategies

A strategy such as always predict that a branch is taken and its converse (always predict that a branch is not taken) are two of the simplest examples of static prediction strategies. A further refinement is to make a prediction based on the type of branch instruction determined by examining the operation code.

In the strategy based on operation code, when a branch instruction with a certain operation code is encountered, a prediction is made as to whether the branch will be taken or not. This decision is predefined in the Instruction Pipeline based on experimental measurements on behaviour of branch instructions. For example experiments show [47], that among the different branch instructions of IBM 370, maximum prediction accuracy is attained if branch instructions with mnemonics B, BR, BAL, BALR, BCT, BXLE, BCR, EX and SVC are always predicted to be taken and all others never taken. Results of experiments carried out on estimating the prediction

accuracy using this strategy have been included in [47] and [57].

2.2.2 Dynamic Prediction Strategies

Dynamic prediction strategies make branch prediction based on past branch history and uses the previous sequence of taken/not taken for each branch to predict whether or not the branch will be taken next time it occurs. Dynamic strategies differ in the amount of branch history they use for the purpose of prediction. The most powerful predictor will be one which uses the entire history of the branch to predict the next choice, but such a predictor is infeasible because of the large number of possible past sequences. Consequently the problem reduces to determining the most desirable amount of history to retain and determine the prediction accuracy for a given amount of history.

The simplest dynamic strategy base their predictions on the most recent branch execution and use an associative memory that contains the address of the n most recent branch instructions together with a bit indicating whether the branch was taken or not taken. The memory is accessed with the address of the branch instruction to be predicted and the taken or not taken bit is used to make the prediction. If a branch instruction is not found in the table, two issues have to be considered. First, in such a case what is the prediction to be made and secondly which table entry should be replaced to make room for the new branch instruction. The first issue is resolved by always taking the branch and as far as replacement strategies are concerned, a replacement strategy such as FIFO or LRU can be used.

A strategy for branch prediction based on four previous occurrences have been proposed by Lee and Smith in [47]. In [47] they have presented results of measurements on the probability of all possible sequences of five taken/not taken events

for conditional branches which is denoted by y_1, y_2, y_3, y_4, y_5 . This data may be used for prediction as follows. If F is the probability that a branch is taken and y_1, y_2, y_3, y_4, y_5 are the events on which F is conditioned, then whenever the probability of $F(y_1, y_2, y_3, y_4, T)$ is greater than $F(y_1, y_2, y_3, y_4, N)$, the branch should be predicted as taken (where y_1, y_2, y_3, y_4 is the sequence of four previous occurrences and T denotes taken while N denotes not taken). According to Lee and Smith the n preceding executions of the branch in question may be remembered in the Branch Target Buffer with n bits. Details of this strategy can be found in [47]. From the results of the experiments, they also found that it may not be necessary to retain a large amount of branch history to get a reasonably good value of prediction accuracy. As observed in [47], the prediction accuracy approaches very close to its maximum value with one, two, or three preceding branches used for prediction. Increasing the amount of history to four or five branches does not seem to add accuracy.

The sole aim of experimenting with different branch prediction strategies is to get a high value of prediction accuracy. Evidently, a high value of prediction accuracy will be able to reduce the performance degradation from branches in the instruction stream. However, presently there exist no means of studying the sensitivity of the performance of an Instruction Pipeline to the prediction accuracy. This aspect is the subject of the next chapter, where the performance of an Instruction Pipeline is analysed to give an expression relating the two.

Chapter Three

Impact of Prediction Accuracy on the Performance of an Instruction Pipeline

As discussed in the previous chapter, a weakness of an Instruction Pipeline is 'bubbles' in the pipeline due to the presence of conditional branch instructions in an instruction stream. Although a great deal of effort has been invested in overcoming the performance degradation due to the presence of conditional branches, its adverse effect cannot be overcome completely. Performance degradation from branches in the instruction stream can be reduced in a number of ways namely loop buffers, multiple instruction streams, prefetch branch target, delayed branch, taken/not taken switch and the branch target buffer. Most of the recent approaches to the branch problem attempt to predict whether or not a branch will be taken. If the prediction is correct, then execution of the correct sequence of instructions continues without delay. But if the prediction is wrong, the initiated instructions have to be 'squashed'. The performance of the Instruction Pipeline therefore depends on the *prediction accuracy* i.e., on the probability of correct branch prediction. In this chapter, the performance of an Instruction Pipeline is analysed to give a quantitative measure of the performance, relating it to the probability of correct branch prediction.

Performance of pipeline processors has been studied by a number of researchers. Chen [8] was the first to suggest the derivation of an expression for Utilisation or Efficiency of a pipeline processor from the space-time relationship. The processing of an instruction stream requires the occupation of equipment space (i.e., pipeline segments) over certain lengths of time. This is represented by the enclosed

area of space-time diagram as in Figure 3.1(a), which shows the actual segment usage involved in the processing of L instructions of a job in a four segment pipeline. In [8], the Efficiency or Utilisation of a pipeline processor in executing a job has been given by the ratio of space-time of job to the space-time area swept by pipeline processor, when all the pipeline segments take the same amount of time for execution, successive instructions are independent and there is a steady flow of instructions through the pipeline. Ramamoorthy and Li [51] have given an expression for Efficiency when segment times are unequal and successive instructions are independent. Both Chen [8] and Ramamoorthy and Li [51] have given expressions for Efficiency on the assumption of a steady flow of instructions through the pipeline, which unfortunately does not hold in practice due to precedence constraints. Baskett and Keller [6] have reported the results of the studies conducted for evaluation of the performance of CRAY-1 computer while Holgate et. al. [24] have given the results of measurements made by hardware monitoring on MU5 system. Ramamoorthy and Wah [53] have studied the degradation in memory utilisation in a pipelined processor due to dependencies in the instruction stream.

In an Instruction Pipeline, the different segments usually take different amounts of time for performing the assigned suboperation and the throughput is governed by the speed of the slowest segment referred to as the 'bottleneck'. Here the expression for Efficiency or Utilisation of pipeline segments is deduced from the space-time or the geometric model for a pipeline with unequal segment times incorporating the effect of conditional branches and the probability of correct branch prediction. Study of the instruction set of different computers reveals that most of them have two types of conditional branch instructions: those which branch to a target address depending on the testing of the condition code set by some previous instruction and those which

Figure 3.1(a) Space-time diagram when processing L instructions ($I_1, I_2, \dots, I_L$) in a four segment pipeline

Figure 3.1(b) Space-time diagram of Figure 3.1(a) recast.

The shaded regions are equal in area

transfer control to some address depending on the outcome of certain computation which it performs. They have different effects on the performance and the analysis takes this into account. To the author's knowledge this is the first time that such an analysis is done. The results given here enable one to study the sensitivity of the performance of a pipeline processor to various factors including prediction accuracy. The simplicity of the analysis followed for obtaining the values of the different parameters makes the methodology attractive.

For the purpose of analysis, it is assumed that the Instruction Pipeline is provided with a fully asynchronous control structure i.e., it allows independent instructions after the conflicting one to continue. This overcomes the problem created by data dependencies among the instructions. Further, the pipeline has multiple arithmetic logic units to avoid the effect of operational dependency. Therefore the performance of the pipeline is degraded only when a branch instruction is encountered. The expression for Efficiency is deduced in terms of the number of conditional branch instructions and the total number of instructions executed. These can be estimated from the run time characteristics of a program to determine its suitability for processing on an Instruction Pipeline.

The methodology followed here is illustrated in section 3.1 by taking the case when successive instructions are independent. In section 3.2, this technique is applied to obtain all the necessary expressions in the more general case. The effect of prediction accuracy is taken into account in the expressions deduced in section 3.3.

3.1 Efficiency with no Precedence Constraint

In this section, an expression for Efficiency is deduced for a pipeline with

s segments, when instructions are independent and segment times are unequal.

Let the segments of the pipeline be $s_1, s_2, s_3, \dots, s_s$.

Time required by segments $s_1, s_2, \dots, s_s$ are $t_1, t_2, t_3, \dots, t_s$ respectively.

The processing of a sequence of L instructions is analysed.

Each instruction occupies a space time $\sum_{i=1}^s t_i$ assuming segment space to be unity.

Space-time area of job consisting of L instructions $= L \sum_{i=1}^s t_i$

Space-time area swept by pipeline processor will include the time lost before the first result is outputted i.e., during initialisation and also during termination of last instruction.

If segment times are equal to τ , then

Space-time lost due to initialisation $= \sum_{j=1}^{s-1} (s-j) \tau$

where $t_1 = t_2 = t_3 = \dots = t_s = \tau$

Space-time lost during termination $= \sum_{j=1}^{s-1} j \tau$

Total space-time lost due to initialisation and termination is by the addition of the above

two expressions $= s(s-1) \tau$

The space-time diagram may be recast for $L > s$, as shown in Figure 3.1(b)

Time required for outputting the first instruction $= \sum_{i=1}^s t_i$

For $L-1$ instructions, the space-time area swept by pipeline processor

$= s(L-1) * \max(t_1, t_2, \dots, t_s)$

$$= s(L-1)t_b$$

where t_b corresponds to the time of the slowest segment of pipeline called the

$$\text{'bottleneck'} = \max(t_1, t_2, \dots, t_s)$$

$$\text{Efficiency } \eta = \frac{\text{space-time of job}}{\text{space-time swept by a pipeline processor}}$$

$$= \frac{L \sum_{i=1}^s t_i}{s \sum_{i=1}^s t_i + s(L-1)t_b}$$

If segment times are equal then $t_1 = t_2 = t_3 = \dots = t_s = \tau$

$$\text{then Efficiency } \eta = \frac{L}{s + L - 1} \text{ as shown by Chen [8]}$$

3.2. Efficiency with Procedural Dependency

The above analysis holds for a continuous excitation of the pipeline assuming no time is lost for satisfying dependencies among instructions, which delays the execution time of the program and results in lower utilisation of the pipeline segments and loss of Efficiency. Maximum penalty is paid when the result of instruction i is used by instruction $i+1$. When segment times are assumed equal to τ , a maximum delay of $(s-1)\tau$ may be introduced into the stream for satisfying a dependency, if conflict is detected in the first stage of the pipeline.

In an Instruction Pipeline instructions are normally executed in sequence

till a branch instruction causes a disruption of the sequence. If the instruction is an unconditional branch, then the dependency can be resolved without much delay and removed by fetching the correct sequence of instructions. But, if it is a conditional branch instruction, data test has to be made for the specified condition. Only then can the correct sequence of instructions be fetched and execution of the instructions has to be held up till the condition is tested and branch path determined. Here the degradation is serious and unavoidable.

As in section 3.1, for simplifying the analysis it is assumed that the Instruction Pipeline is equipped with a fully asynchronous control structure which allows independent instructions after the conflicting one to continue and the pipeline has a number of arithmetic logic units so that there is no delay due to contention for the execution unit. It is assumed that no computation is required for branch address calculation such as addition of displacement to a base or index register. The instruction fetch stage may have a buffer for storing instructions in order to reduce main memory access and has logic for prefetching the branch target. That is, when a branch is recognised, a special mechanism calculates and prefetches the target of the branch. Thus, if the branch is found to be taken, the target instruction is loaded immediately into the instruction decode stage of the pipeline, with no additional delay for fetching the instruction.

Conditional branch instructions in an instruction stream are of two types: those which branch to a target address depending on the testing of the condition code set by some previous instruction and those which transfer control to some address depending on the outcome of certain computation which it performs. For example in IBM 360/370 system [29], branch instructions with pmonics BC and BCR fall into the first category and BXLE, BXH, BCT, BCTR are of the second type. Let the number of

instructions in these two categories be d_1 and d_2 respectively. Let d_3 be the number of unconditional branch instructions in an instruction stream among the total number of instructions L .

τ is the segment time.

If N be the number of instructions in between the instruction setting a condition code and the instruction which tests this result then for $N \geq s - 1$, no delay is encountered.

For $N < s - 1$, the dependency causes a delay of $(s - 1 - N) \tau$ to the execution time of the program.

Conditional branch instructions of the first type (i.e., d_1) and Unconditional branch instructions (i.e., d_3) are assumed to occupy no segment space.

Then, space-time of job = $(L - d_1 - d_3) s \tau$

Space-time lost waiting for condition code generation for one conditional branch instruction = $(s - 1 - N) s \tau$

Total space-time lost waiting for condition code generation due to d_1 branch instructions would be:

$$d_1(s - 1) s \tau - \left(\sum_{i=1}^{d_1} N_i \right) s \tau = d_1 [s - 1 - N_{av}] s \tau, \text{ where } N_i \text{ is the number of instructions in between the instruction setting a condition code and the branch instruction testing it for the } i \text{th branch instruction and } N_{av} = \text{Average value of } N_i \\ = \sum_{i=1}^{d_1} N_i / d_1$$

Conditional branch instructions of the second type cause serious

degradation in performance as successive instructions cannot be initiated till these branch instructions finish the computation and determine the target address. If the target instruction is present in the instruction buffer in the Instruction Fetch stage, then memory access time may be neglected. For each of this type of instruction, the execution of the instruction stream is delayed by $(s - 1) \tau$ time units.

Hence space-time lost due to d_2 conditional branch instructions is $d_2 (s - 1) s \tau$.

It has already been shown in section 3.1 that the space-time lost due to initialisation and termination operation = $s (s - 1) \tau$

Total space-time area swept by pipeline processor =

$$s (s - 1) \tau + d_1 (s - 1 - N_{av}) s \tau + d_2 (s - 1) s \tau + (L - d_1 - d_3) s \tau$$

$$\begin{aligned} \text{Therefore, Efficiency } \eta &= \frac{(L - d_1 - d_3) s \tau}{(L - d_1 - d_3) s \tau + s (s - 1) \tau + d_1 (s - 1 - N_{av}) s \tau + d_2 (s - 1) s \tau} \\ &= \frac{(L - d_1 - d_3)}{(L - d_1 - d_3) + (s - 1) + d_1 (s - 1 - N_{av}) + d_2 (s - 1)} \end{aligned} \quad (1)$$

When segment times are unequal the following holds:

$$\text{Space-time of job} = (L - d_1 - d_3) \sum_{i=1}^s t_i$$

Space-time area swept by pipeline processor is obtained from the recast space-time diagram in the same way as discussed in section 3.1 and is given by

$$s \left[\sum_{i=1}^s t_i + \{ d_1 (s - 1 - N_{av}) + (L - d_1 - d_3 - 1) + d_2 (s - 1) \} t_b \right]$$

$$\text{Therefore, } \eta = \frac{(L - d_1 - d_3) \sum_{i=1}^s t_i}{s \sum_{i=1}^s t_i + s \{d_1 (s-1 - N_{av}) + d_2 (s-1) + (L - d_1 - d_3 - 1)\} t_b} \quad (2)$$

3.3 Effect of Prediction Accuracy

Review of the mechanisms that have been proposed for reducing the delay due to presence of conditional branch instructions in an instruction stream reveals that an acceptable solution has to be based on predicting whether or not a branch will be taken and conditionally forwarding the next instruction to the pipeline either from the target address or from the next sequential address depending upon the guess. If the prediction is correct, then execution continues instantaneously without time delay. To take the effect of prediction into account the expression for Efficiency deduced above is modified, as correct prediction will reduce the waiting time. This is done as follows.

If p is the probability that the branch prediction is correct (which is referred to as the prediction accuracy), then the value of the Efficiency can be obtained by substituting $d_1 * (1 - p)$ and $d_2 * (1 - p)$ for d_1 and d_2 in the expressions (1) and (2) deduced above.

In this case, the expression for η is as follows:

$$\eta = \frac{\{L - d_1(1-p) - d_3\} \sum_{i=1}^s t_i}{s \sum_{i=1}^s t_i + s \{d_1(1-p)*(s-1 - N_{av}) + d_2(1-p)*(s-1) + L - d_1(1-p) - d_3 - 1\} t_b} \quad (3)$$

When segment times are equal, the value of η becomes,

$$\eta = \frac{\{L - d_1(1-p) - d_3\}}{s + d_1(1-p)*(s - 1 - N_{av}) + d_2(1-p)*(s - 1) + L - d_1(1-p) - d_3 - 1} \quad (4)$$

The expressions (3) and (4) deduced above enable one to study the sensitivity of the performance of an Instruction Pipeline to the prediction accuracy. The variation of η with p may be obtained from this expression for different values of d_1, d_2, d_3, L and N_{av} for a pipeline with s number of segments. Values of d_1, d_2, d_3, L and N_{av} can be obtained by dynamic measurement on application programs. In Figure 3.2, η has been plotted against p for some typical values of d_1, d_2, d_3, L and N_{av} to show the variation of Efficiency with the prediction accuracy for a five segment pipeline with equal segment times. The results plotted in the graph of Figure 3.2 indicate that a prediction accuracy of 0.8 can give reasonably good values of utilisation of pipeline segments.

Figure 3.2 Variation of *Efficiency* with *Prediction Accuracy* for a Pipeline with equal segment times.

Chapter Four

Arithmetic Pipeline

The technique of pipelining has found widespread application in the realisation of some complex arithmetic operations. Such systems called Arithmetic Pipelines have been used for example, in IBM 360/91 [4] and CDC Star 100 [10], for the high speed execution of algorithms for floating point arithmetic.

A number of studies have been conducted on pipelining arithmetic algorithms. Some of these have been discussed in [40]. Hallin and Flynn [19] had studied pipelining two addition and three multiplication algorithms using a fast register called Earle Latch for isolation between pipeline stages. Jump and Ahuja [32] have discussed the application of pipelining to the class of problems which can be realised as two dimensional cellular arrays, where each cell is a combinational logic circuit connected to its four nearest neighbours. Their analysis is based on an n stage pipeline obtained by partitioning a non-pipelined system into n smaller subsystems and then adding registers between stages. The additional cost of such a pipelined system is due to the registers added. They have studied different ways of pipelining an algorithm from the viewpoint of cost effectiveness. Chen has discussed in [9] algorithms for binary addition, multiplication, and division, which are suitable for pipeline execution. In [39], Kogge has proposed a method for pipelined realisation of recurrence functions.

Pipelines for arithmetic processing such as those used for multiplication and division are characterised by the following attributes: the speed of each segment is fixed and the execution process requires internal looping i.e., the results are fed back as inputs. This gives rise to the sequencing control problem i.e., determine expedient

moments at which to introduce new input from an external source so that there is no collision (which takes place when two computations attempt to use the same segment). The problem is discussed below.

4.1 Collision Prevention and Reservation Table

For initiating computations without collisions, a Reservation Table is used to describe the flow of computation through the segments of a pipeline. The problem is discussed here with the help of an example of the pipeline shown in Figure 4.1(a). The traversal of operands through the pipe is shown in the Reservation Table of Figure 4.1(b).

Each segment S_i requires one unit of time for processing. The computational sequence requires the passage of the operands in the order $S_1, S_2, S_3, S_4, S_1, S_2$. In the Reservation Table, an X is placed at the intersection of a segment's row with columns corresponding to each time unit relative to the initiation during which that segment is used by the computation. Consider the case in which a computation has just begun. The Reservation Table is analysed to determine at which future times a new computation may be initiated without causing a collision. The minimum time that must elapse between the initiation of two computations is called *latency*. One way to determine whether two computations may be initiated K units of time apart is to superimpose the reservation on itself offset by K units of time. If an X falls on top of another, then a collision will occur in that segment and K is a *forbidden latency*. Otherwise no collision occurs, and K is an *allowable latency*. Thus, if any pair of X s in any row is K units of time apart, then K is a forbidden latency. Therefore it is simple to construct a *forbidden*

(a) A Pipeline

Segment	t_1	t_2	t_3	t_4	t_5	t_6	t_7
S_1	X				X		
S_2		X				X	
S_3			X				
S_4				X			

(b) Reservation Table

Figure 4.1 A Pipeline and the Reservation Table for a computational process

list F , which is a list of all all forbidden latencies for the particular Reservation Table.

The *minimum constant latency* is the smallest number L such that any number of computations can be initiated L time units apart without collisions. The minimum constant latency L can be determined from the forbidden list F using the fact that L is the smallest integer $n \geq 1$ such that kn is not in F for any integer $k \geq 1$. By allowing variable periods between initiation of computations, it may be possible to determine a scheduling strategy which yields a lower average latency i.e., higher throughput, than if the minimum constant latency is used. A strategy that minimises the average latency is based on the use of collision vector.

If in the forbidden list, n is the largest element, it is possible to construct the *collision vector*, which is a binary vector of n bits from 1 to n . Bit i in this vector is a 1 if and only if it is an element of the forbidden list. Thus, if the collision vector is $C = c_n c_{n-1} \dots c_2 c_1$, then $c_i = 1$ if i is an element of the forbidden list; otherwise $c_i = 0$. For the example of Figure 4.1(a), the forbidden list is (4, 4) and $n = 4$, making the collision vector 1000.

By the use of collision vector, a simple control mechanism can be used to avoid collisions. Before initiating a new computation, the collision vector is checked to see if there are zeros in every location corresponding to the number of time units that have elapsed since each previous computation was initiated. Davidson [13] devised an ingenious shift register for checking this requirement. However, this is not discussed here.

4.2 Types of Arithmetic Pipeline

Usually in an Arithmetic Pipeline each stage is designed to perform a particular suboperation and the computation performed by the pipeline is determined by the interconnection between the stages. This results in the Arithmetic Pipeline to be dedicated to certain types of computation. An Arithmetic Pipeline with a fixed and dedicated function is called a *unifunctional* Pipeline. An example of this is one, which performs only floating point addition. An unifunctional Arithmetic Pipeline has limited applicability. On the other hand, a *multifunctional pipeline* provided with capability for reconfiguration may perform different functions, either at different times or at the same time by interconnecting different stages in the pipeline. Accordingly, a multifunctional pipeline is said to be static or dynamic. It is called *Static*, if at any time instant, only one configuration is active. CDC Star 100 [10] and TI ASC [62] have static multifunctional pipelines which can reconfigure under microcode control. *Dynamic* multifunctional pipelines permit overlapped processing among several active configurations simultaneously. Throughput may be enhanced but it requires complex control and sequencing method.

4.2.1 Static Multifunctional Pipeline

By definition, a static multifunctional pipeline can perform different functions at different times under program control or firmware control. For understanding the issues involved in the design of static multifunctional pipelines, two examples are discussed.

A four function pipeline proposed by Kamal et. al. [33] in 1974 is a good

example of a static multifunctional pipeline. It can perform multiply, divide, squaring and square root operations. Two types of cells (called A cell and K cell) are used in its construction. The two cell types are specified in Figure 4.2 [40] by boolean equations.

The A cell (Figure 4.2(a)) is a combined adder/subtractor with two control inputs, four data inputs and four data outputs. The data outputs S, D and E are latched. The control input X indicates whether or not the data input B is to be left alone or complemented. A 0 on control input F passes the input A to the output S; a 1 adds A to B as modified by X. The data outputs D and E pass various logical functions of B and C as selected by F.

The schematic design of the four function pipeline with A cells, K cells and interface latches is shown in Figure 4.3 [40]. Each stage of the pipeline consists of a horizontal line of computation cells connected as a ripple adder/subtractor. There are four sets of data inputs A, B, C and P and one set of outputs S. Depending on which operation is being performed, a different set of outputs are used. The X and F controls for each stage are unique and are derived from the control cell (i.e., K cell). This cell is driven by delayed versions of the X and P inputs.

Table 4.1[40] lists the values applied to the pipeline's input to initiate each of the four basic operations. For example, for multiplication the B's are one input and the P's the other and X is set to zero. At each stage, one of the P bits feeds the control cell. If the bit is a 1, F is a 1 and the B input is added into the current sum in the proper bit position. If the P bit is 0, the current sum is passed to the next stage unaltered.

Results of multiply, squaring and remainders of divide and square root are generated at the bottom stage of the pipeline. Quotient bits and root bits are generated at the left end of each pipeline stage in a sequential manner.

(a) Multifunction computation cell

(b) Associated control cell

Figure 4.2 The cells and its logic equations

Figure 4.3 A four function Arithmetic Pipeline

Table 4.1 Inputs required for the four functions

Operation	Required Input
$(A_1 \dots A_6) + (B_1 \dots B_4) \times (P_2 \dots P_4)$	$X=0=P_1$ $B_i=0$ for $i > 4$ $C_i=B_i$
$(A_1 \dots A_6) + (B_1 \dots B_4)$	$X=1$ $B_i=0$ for $i > 4$ $C_i=B_i$ $P_i=0$
$(P_2 \dots P_4)^2$	$X=0=P_1$ $B_i=0$ for $i > 4$ $C_i=B_i=P_i$ for $i \leq 4$
$\sqrt{(A_1 \dots A_8)}$	$X=1$ $P_i=0$ $B_1 C_1=00$ $B_2 C_2=01$ $B_i C_i=10$ for $i \geq 3$

The TI ASC Arithmetic Pipeline consisting of eight stages is another example of static multifunctional pipeline. This pipeline can perform either fixed point or floating point arithmetic functions and many logical shifting operations over scalar and vector operands of lengths 16, 32 or 64 bits. Different arithmetic logic instructions are allowed to use different connecting paths through the pipeline. Figure 4.4 shows four interconnection patterns of the ASC pipeline for the evaluation of the functions: fixed point add, floating point add, fixed point multiply, and floating point vector dot product. The receiver and output stages are used by all instructions.

In TI ASC, different paths are assumed under microprogram control for the execution of different instructions. All path control information is stored in a ROM, which is accessed at the initiation of an instruction. For example, configuration for floating point addition requires four ROM words for its path interconnection information. The instruction execution logic has to access the ROM for control signals. The ROM words for a floating point add may be located at 100, 101, 102 and 103 while words for a floating point subtract could be located at 200, 101, 102 and 103. The common ROM words (101, 102, 103) used by both the arithmetic operations represent similar suboperations contained in these two instructions. The starting ROM address is supplied by the instruction execution logic directly after the decoding of the instruction.

4.2.2 Dynamic Multifunctional Pipeline

A dynamic pipeline is one which allows several configurations to be active simultaneously. For example, a dynamic Arithmetic Pipeline can perform addition and multiplication at the same time. Tremendous control overhead and interconnection complexity is involved for achieving this and none of the existing commercial pipeline

Figure 4.4 Four functional configurations in the TI-ASC Arithmetic Pipeline

processors have this dynamic capability.

A dynamic pipeline has to initiate tasks from different Reservation Tables corresponding to the different computations which have to be performed simultaneously in the same pipeline. Control problem arises because as each of the pipeline segments is assigned to perform a particular subfunction, different pipeline segments have to be interconnected simultaneously for executing different arithmetic computations at the same time. The problem has been discussed in [26]. The control complexity is reduced in an Arithmetic Pipeline designed on the basis of universality of a pipeline stage i.e., any stage is capable of performing any arithmetic suboperation. This aspect is discussed further in the next chapter, where a design of an Adaptable Pipeline is proposed based on this philosophy.

4.3 Vector Processors

Pipelining finds a natural application in vector processing. If the same operation is performed on all the elements of independent vectors then a continuous initiation of the pipeline can be obtained. To facilitate processing of vectors, pipeline systems have been built with vector instructions in its instruction set and having special hardware for fetching consecutive elements of the vectors for execution. Such pipeline systems have been termed Vector Processors. CDC Star 100 [10], CRAY-1 [12], TI-ASC [62], are prominent examples of systems with additional hardware for vector processing.

Vector Processing generally involves two levels of pipeline action. In the first level, a vector instruction is fetched, decoded and the necessary control paths connected before the needed elements of the vector are fetched from consecutive storage

locations over a specified address range. In the second level, the arithmetic execution unit carries out the specified operation on these elements, normally being supervised by a control ROM. Vector processing has associated with it some overhead. In order for such processing to be worthwhile, the 'setup time' (i.e., the time taken to structure the pipeline and prepare the streams of vector operands) and the 'flush time' (i.e., time to empty the pipe at the conclusion of the operations and set the termination conditions) must be small.

Analysis reveals [52] that the length of the vector field must be atleast ten for vector processing to be advantageous over processing in a sequential pipeline.

In recent years, substantial efforts are being exerted on developing high level languages with parallel constructs to facilitate Vector Processing. The language FORTRAN has been extended to support vector operation by the inclusion of special primitives such as vector addition and vector multiplication. Examples of such extensions are LRL-tran for Star 100 computer and Texas Instruments ASC FORTRAN for their TI ASC computer [52]. Two vector processing languages have been proposed recently: one is Vectran and the other is Actus [50]. The alternative to using parallel processing languages is the installation of intelligent compilers with optimisation procedures for detecting vector operations in sequential program statements and translating them into powerful vector arithmetic instructions. A vector instruction usually indicates the type of operation to be performed (e.g. floating point add), the beginning address of each vector and the amount by which this address has to be incremented to find the successive vector elements to be processed. The design of vectorising compilers has been discussed in [25] and [42].

In the next chapter, the design of an Adaptable Pipeline is described. Its

design philosophy is different from that of a conventional Arithmetic Pipeline. It has multifarious processing capabilities and can simultaneously perform different types of arithmetic functions.

Chapter Five

Design of an Adaptable Pipeline based on Transputers

As already discussed in the previous chapter, an Arithmetic Pipeline tends to be dedicated to a particular computational process depending on the operation performed by each of the segments of the pipeline and the interconnection between them. Multifunctional pipelines are capable of performing a number of functions but are usually static i.e., at any time instant only one configuration is active. Throughput can be much higher in a dynamic multifunctional pipeline but because of control problem none of the existing multifunctional pipeline systems have achieved this dynamic capability. The control complexity can be reduced by designing an Adaptable Pipeline based on the universality of a pipeline stage as it eliminates any need for hardware reconfiguration. In this chapter, the design of an Adaptable Pipeline is described which is based on VLSI processing elements called Transputers [63].

In such a pipeline system it is possible to execute various types of computational processes at the same time. The computational processes which are capable of simultaneous execution in an Adaptable Pipeline are:

- a complex arithmetic operation like floating point addition, floating point multiplication
- a vector instruction
- a task, which is decomposed into subtasks each of which consists of a sequence of instructions. The same data stream is acted on by a cascade of subtasks, one after another in a pipelined mode. The data stream passes through the first stage of the pipeline and the results are sent to the second stage. The second stage then passes the refined results to the third stage and so on. This has been referred to as processor

pipelining in [26].

Further, the Adaptable Pipeline is capable of switching its resources into different states, where each state is specified by the number of concurrent pipelines and the number of stages in each pipeline. For example in one state, the entire resource may go into the formation of three pipelines, each containing five stages. In another state, the system may form two pipelines with four stages each and two pipelines with three stages each. Such dynamic redistribution of resources facilitates the concurrent execution of instruction streams and also permits the organisation of vector chaining [52].

The concept of adaptable pipelining, was the basis of an earlier architecture proposed by Kartashev and Kartashev in 1979 and described in [35] [36]. Their design is based on special purpose LSI chips called Dynamic Computer groups (DC groups) [34] [37]. The pipeline designed by them can be used only when the computational process does not require internal looping i.e., there is no necessity for the results to be fed back as inputs. The design described here overcomes this drawback. Besides making provision for internal looping, the present design based on Transputers puts greater emphasis on limiting the complexity of the interconnection switches and on increasing the versatility of the pipeline stages, thereby making the Adaptable Pipeline system useful in a wide range of processing applications.

The hardware organisation of the Adaptable Pipeline is described in section 5.1 and its mode of operation is illustrated in section 5.3.

5.1 Hardware Organisation of the Adaptable Pipeline

Each stage of the Adaptable Pipeline is built from a Transputer, whose salient architectural features are given in [63]. Figure 5.1 shows a Transputer and the

pins which will be frequently referred to in the present discussion.

A linear array of s , 32 bit Transputers (the T424 chips [30]) forms the consecutive stages of the Adaptable Pipeline (as shown in Figure 5.2). They are named $T_0, T_1, T_2, \dots, T_{s-1}$ and have addresses $0, 1, 2, \dots, s-1$. The Transputer array is connected to a computer Co , which acts as a supervisor. Computer Co stores instructions and data in its memory Mo and fetches them to the first stage of a pipeline. An instruction is initiated for execution from the first stage of the pipeline. After being processed by the first stage, the results and the instruction are sent to the second stage. The second stage then executes the operation assigned to it and passes the refined result and the instruction to the third stage for further processing. The process continues till the instruction is completely executed. After an instruction has been completely executed; the results are sent back to Co for storage in Mo .

The instruction sent by Co for processing in the Adaptable Pipeline has the format given in the next section. It has three fields. The operation performed by a particular stage of the pipeline is determined by two of these fields D and w . An instruction as it reaches a pipeline stage is interpreted by a set of programs and control is transferred to the appropriate routine on the basis of the contents of these fields. This routine has all the necessary statements for performing the desired arithmetic operation. A computational process partitioned into n subprocesses passes through $n+1$ consecutive stages of the pipeline: the first n stages perform the necessary arithmetic operations while the result is sent back to Co by the $n+1$ th stage. This technique allows one to perform an adaptation to the number of pipeline stages that is required for the complete execution of a single instruction without any necessity for bypassing unneeded stages thereby avoiding conflict resolution problems associated with such bypassing.

P : Processor
 M : Memory
 LI : Link Interface

Figure 5.1 Links of a Transputer

Figure 5.2 Hardware Organisation of an Adaptable Pipeline

As already stated, an Adaptable Pipeline has the capability to form many parallel pipelines of variable lengths. For incorporating this feature, the link Lin0 of all the Transputers is connected to Co. This ensures that any Transputer may receive instructions from Co thereby becoming the first stage of a pipeline.

After the completion of the execution of an instruction, the final result is sent back to Co for it to be stored in Mo. An instruction implements a sequence of w operations and ends its execution in stage T_w , and the value of w varies with different instructions. That is, any pipeline stage can be the end stage of some instruction. Therefore, every pipeline stage must be capable of sending computational results to the main memory through Co. For this purpose, the link Lout0 of all the Transputers forming the different stages is connected to the supervisory computer Co. Besides dispatching instructions to all the first stages of the pipelines formed and receiving the results for storage into memory, Co has the task of monitoring the operation of the entire system. All these objectives are satisfied by connecting the links Lin0, Lout0 of the Transputers to Co.

7 For each Transputer T_i , its two links namely the Lin1, Lout1 are used to connect it to its successive pipeline stage. They are used for right transfer of instruction and data from T_i to T_{i+1} . Although not shown in Figure 5.2, Links Lin1, Lout1 of the Transputer at the end of the array (i.e., T_{s-1}) are connected to the Transputer at the beginning of the array (i.e., T_0). This is done to enable the feedback of computational results from one stage to a stage preceding it. For the i th Transputer, instruction and data shifted from T_{i-1} are received in T_i in line Lin3, while acknowledge signal is sent in line Lout3. As each stage of the pipeline needs to have the capability of sending

temporary results to other stages, one pair of links Lin2, Lout2 go to the Routing Network through which the results are sent to the appropriate stage. If the result of an operation in stage T_i is a future temporary result for another stage T_j , then it is sent through the Routing Network to that stage and written into its local memory.

Each pipeline stage may send its computational result to any stage, through the Routing Network. A Transputer T_i intending to send a temporary result to T_j sends a code d_j to the Routing Network. The code activates the appropriate switches to establish the desired connection between the sending Transputer (T_i) and the addressed Transputer (T_j) through the serial links. After establishment of the connection, the bits constituting the code d_j are allowed to proceed to the addressed Transputer T_j through its serial links and an acknowledge signal is sent back from it to the sending Transputer (T_i). After the acknowledge signal is received, T_i can start sending the data value of the temporary result. If s is the total number of stages in the Adaptable Pipeline, then the number of bits required in d_j for encoding the addresses of all the stages is $\log_2 s$.

Each pipeline stage T_i uses its 4K local memory for storing data and the programs necessary for the interpretation of the instructions sent to it either by Co or the previous stage in the pipeline. It also stores temporary results required by processor in Transputer T_i . Each Transputer has an external memory connected to it. This is used when the Transputer's local memory is not sufficient to hold all the programs and data required for the operation of that stage.

5.2 Format of an Instruction

The instruction sent from Co for high speed execution in the Adaptable Pipeline has the format shown in Figure 5.3 and has three fields. In order to specify the operation, it is necessary that each instruction contain an operation code (D). However, when the same D propagates through the pipeline stages, it will activate the same operation in each stage. Selective activation of an operation assigned to stage T_i by an instruction is achieved with the help of another field w.

Figure 5.3 Instruction Format

The field w gives the number of consecutive pipeline stages necessary for the execution of that instruction. Its initial value equals the number of consecutive operations required for the realisation of that instruction. When an instruction propagates through a stage of the pipeline, the value of w is decremented by 1 and the instruction is passed on to the next stage with this modified value of w. When an instruction reaches a pipeline stage for execution, the contents of the field w give the number of stages remaining to be traversed before completion of execution of that instruction. A value 0 in the w field implies that the instruction has been completely executed.

The size of the field w is $\log_2 F$ bits, where F is the maximum number of stages of the Adaptable Pipeline required for the execution of any computational process.

For executing computational processes which do not require looping, i.e., when the results of a stage do not have to be fed back as inputs to a previous stage, the value of F can be taken to be equal to the maximal length of the Adaptable Pipeline. When a computational process requires looping, the computation of F is more complicated. This is illustrated with an example.

Consider a computational process (say an instruction) whose usage of different stages of a pipeline is shown in Figure 5.4(a). In this discussion, such a figure is referred to as *Data Flow Graph*. The numbers preceded by # sign inside the circles denote the different types of arithmetic operations performed in the different stages of a four segment pipeline. From the figure, it is clear that the computational process uses stages 2 and 3, two times i.e., the sequence of traversal of the stages for the complete execution of the instruction is 1 2 3 2 3 4. Now consider the execution of the same instruction in an Adaptable Pipeline whose design is based on the universality of a pipeline stage i.e., any stage can perform any arithmetic operation. The same computation may be performed in the Adaptable Pipeline by using the different stages as shown in Figure 5.4(b). That is, instead of data flowing in a small loop between stages 2 and 3, there is only one large loop from the last stage to the first one. Therefore, the total number of stages traversed for executing this instruction is 6. The value of F is taken as the maximum of all these values required by different computational processes and size of field w is calculated therefrom. The corresponding reservation table is shown in Figure 5.4(c).

The operation performed by a stage is determined by the values of D and w . The instruction reaching a stage is interpreted by a program contained in that stage and perform the operation specified by the combination of D and w . The program to which control is transferred for performing the necessary arithmetic operation is

(a) Original Data Flow Graph

(b) Modified Data Flow Graph

Segment	Time					
	t1	t2	t3	t4	t5	t6
1	X				X	
2		X				X
3			X			
4				X		

(c) Reservation Table when executed in the Adaptable Pipeline

Figure 5.4 Data Flow Graph of a computational process and its execution in the Adaptable Pipeline

determined by the contents of the fields D and w . The instruction after interpretation by a stage i is passed on to the next stage $i+1$, which performs another arithmetic operation on the results obtained in the previous stage. An instruction passes from one consecutive stage to another till the value of w becomes 0. When any stage i of the pipeline, receives an instruction with 0 in the w field, it immediately knows that the instruction has been completely executed and the results have to be sent back to Co .

In addition to the adaptation codes D and w , each instruction has a field containing the Process Identification Number (PIN). Each computational process (for example an instruction for the execution of a complex arithmetic operation) to be executed in the pipeline is assigned a Process Identification Number by the supervisory computer Co , when it is sent to the first stage of the pipeline. Since a pipeline with s stages can have at most s different instructions residing in its different stages, the values of PIN range from 0 to $s-1$ and the number of bits b required in the PIN field has to satisfy the relation $b = \log_2 s$. For example, a 16 stage pipeline can have PIN values from 0 to 15. Therefore, the instruction format for such a pipeline needs to have 4 bits in the PIN field. Whenever an instruction is dispatched by Co for processing in the Adaptable Pipeline, a number between 0 and $s-1$ is assigned to it in this field such that this number is different from the PIN values of all the instructions currently residing in the pipeline. The contents of the PIN field serve a number of purposes. For example, the data value which is sent from one stage to another is always preceded by the value of PIN, to identify the computational process to which it belongs while the program which interprets the operation code of an instruction is referred to by the value of D and w . Since an instruction has fields containing both PIN and D values, the correspondence between the interpretive program and the variables can be established easily.

5.3 Operation of the Adaptable Pipeline

It has already been stated, that an Adaptable Pipeline designed with Transputers has multifarious processing capabilities. In this section, its mode of operation is illustrated when applied to the high speed execution of a complex arithmetic operation (say an instruction for floating point arithmetic) which has been partitioned into suboperations for execution in different stages of the pipeline. The suboperations to be performed by the different stages are previously determined and the Transputer of each stage is pre-loaded with all software necessary for the purpose. It contains all the programs necessary for the execution of the suboperations of different computational processes that may be assigned to it to perform. Each stage selects the particular program needed for performing a suboperation of an instruction on the basis of the contents of the fields D and w of the instruction.

For the purpose of illustration, it is assumed that the system forms a single pipeline with 16 stages and the stages of the pipeline are T_0 to T_{15} , having addresses 0, 1, 2, ..., 15. The transputers forming the stages are interconnected through the serial links as discussed in section 5.1.

Co, the supervisory computer sends instructions to the first stage of the pipeline. Since it is assumed here that the system forms only one pipeline, the first stage is at T_0 . The instructions sent to T_0 have the format given in section 5.2. After sending the instruction, the Supervisory Computer Co sends the operands required for the execution of this instruction. This is done in the same way as sending the instruction except that before sending the bit values of the operands, the value of the PIN is also sent to T_0 . This protocol enables the Transputer to identify the computational process to

which the operands belong. The operands received from Co are stored in the workspace of the process whose name corresponds to the value in PIN. In the terminology of Transputers, the workspace denotes an area of on-chip memory where local variables are kept. After T_0 receives the instruction and data, the instruction is interpreted to determine the type of suboperation the stage has to perform and control is transferred to the appropriate routine. These routines are stored as processes (in the terminology of OCCAM [31]) with their names corresponding to values in the D and w fields. When control is transferred to a process for performing the assigned suboperation, the variables needed by this process are fetched from the workspace of the process whose name corresponds to the value in the PIN field of the instruction.

When the arithmetic operation assigned to a stage has been performed, the value of w is decremented by 1. The instruction with this modified value of w is then sent to the next stage for further processing together with the result(s) of the arithmetic operation. The process continues till the value of w becomes 0.

When a stage receives an instruction with a value 0 in its w field, it immediately recognises that the instruction has been completely executed and its results have to be sent to Co. When a stage intends to transmit a result to the supervisory computer Co, it first sends the value of the PIN on the serial link and then the bits corresponding to the result are sent. This protocol enables the supervisory computer to establish a correspondence between the instruction sent for processing on the Adaptable Pipeline and the result(s) obtained after the completion of processing.

On the other hand, consider the execution of an instruction whose value in the field w is not 0 even when it reaches the last stage of the pipeline. It becomes evident immediately that the execution of the instruction is not yet over. In that case, the instruction and the results are sent to the first stage of the pipeline for continuation

of processing. This frequently happens when the Adaptable Pipeline is used for execution of computational processes which require traversal through a large number of stages. For example, instructions for floating point arithmetic generally involve looping. When executing such instructions in the Adaptable Pipeline, the inner loops in its data flow graph are replaced by one outer loop as discussed in section 5.2 and the complete execution of such instructions may involve traversing the entire length of the pipeline several times.

If after the end of some suboperation, a stage i intends to send some temporary result(s) to another stage j , where $j \neq i+1$, the Transputer in stage i first sends the identification code d_j of that stage to the Routing Network. The Routing Network establishes the connection between the serial links corresponding to i and j and data transmission from T_i to T_j takes place. Before sending the actual value of the data, the value of PIN is also transmitted to identify the computational process to which it belongs.

Determination of the expedient moments at which to initiate an instruction for processing in the adaptable pipeline is done by C_0 , from an analysis of the reservation tables [40] belonging to different computational processes. This aspect is discussed in section 5.4.1.

5.4 Organisation of a Pipeline Stage

To gain most benefit from the Transputer architecture, the whole system needs to be programmed in OCCAM language [31]. OCCAM is based on the concept of processes and channels and the software for the whole system can be designed in terms

of these. A process performs a sequence of actions and terminates. It communicates with other processes along point to point channels. In this section, the software organisation of a pipeline stage is discussed in terms of the OCCAM model of processes and channels.

OCCAM has the advantage that concurrent processes are automatically synchronised. If a channel is used for input in one process and output in another, communication takes place when both the inputting and outputting processes are ready. The processes then proceed and the value to be output is copied from the outputting process to the inputting process. Therefore, this takes care of the difference in execution times of different stages.

The interconnection of the links between the successive Transputers has already been discussed in section 5.1. Each of these unidirectional links implements an OCCAM channel. Each Transputer has six channels which go in the directions shown in Figure 5.5.

Figure 5.5 Channel Organisation

The links Lin0, Lout0 constitutes Channel 0 (Ch0) and Channel 1 (Ch1) respectively. Link Lout1 from Transputer T_i to T_{i+1} form Channel 2 (Ch2). Instructions and data enter T_i from T_{i-1} through the link Lin3 constituting Channel 5 (Ch5). After the execution of an arithmetic operation, a pipeline stage may need to send the results of the computation not only to its next stage but to another stage T_j through the link Lout2 which forms Channel 4 (Ch4). Data enters the Transputer T_i from other stages via the Routing Network through link Lin2, which constitutes the Channel 3 (Ch3).

The operation of a stage is described by means of a flow chart. The sequence of operations followed is the same for all the stages except when it is the first stage of the pipeline. In that case the flow chart is slightly different and is described in section 5.4.1.

The flow chart in Figure 5.6 gives the sequence of operations followed by a stage when it is not the first stage of the pipeline. The transputer in each stage of the pipeline can get inputs from two channels: Ch5 and Ch3. Each transputer should be programmed to be able to input from these two channels concurrently. As different pipeline stages require different amounts of time for processing, for each stage the time of arrival of data in Ch3 is independent of the time of arrival of instruction and data in Ch5.

As already discussed, an instruction as it reaches a stage is analysed and control is transferred to a routine depending on the values in the fields D and w in the incoming instruction. Besides statements for executing the assigned suboperation, this routine contain processes for sending the instruction and data to other stages. The statements in the output process have to take into account the protocol required to be

Figure 5.6 Flow of operations in a pipeline stage

followed for the purpose. For example, if the Transputer in stage T_i intends to send a data value to another Transputer T_j through the Routing Network, then the output process must contain statements for first outputting the value d_j . This is followed by statements for transmitting the PIN value of the instruction to which the data belongs. Finally T_i transmits the bits giving the value of the data.

5.4.1 Organisation of the First Stage

The sequence of operations followed by the first stage of the pipeline is different from other stages. An instruction is sent by Co to the first stage from where it starts traversing the consecutive stages of the pipeline till completion of its execution. However, Co cannot send instructions to the first stage continuously. This is because execution of certain instructions may require traversing the entire length of the pipeline many times and the time of their visit to the first stage may coincide with the initiation of a new instruction. To avoid such collisions, the time of initiation of instructions have to be determined from an analysis of the reservation tables belonging to different computational processes. A method of initiating new instructions to the pipeline without collisions is now discussed.

As exemplified in section 5.2, the modification of the data flow graphs of computational processes by replacing small inner loops with one outer loop reduces non-linearities in the flow of data. This simplifies the reservation tables of computational processes. For example, the reservation table for the computational process of Figure 5.4(a) when executed in the Adaptable Pipeline is as given in Figure 5.4(c). From the reservation table, it can be seen that the forbidden list [40] of all computational processes

contain only one number which is equal to the number of stages in the pipeline. This makes it easier for Co to analyse the reservation tables of different computational processes and to determine the time of initiation of the different instructions.

It is assumed that after analysis of the reservation tables of different computational processes, Co determines a value 'm' which specifies the number of instructions which must be executed by the first stage before a new instruction can be initiated. As shown in the flow chart of Fig. 5.7, after the execution of every instruction, the value of m is decremented. If on testing it is found that the value of 'm' is not 0, then the first stage inputs from Ch3 i.e., it inputs the instruction and data coming from the last stage of the pipeline. Otherwise it inputs the contents of Ch0 which is a new instruction sent by Co.

It should be noted that the system proposed here is capable of forming many parallel pipelines. Therefore any stage may become the first stage of a pipeline. This necessitates that all the stages be equipped with programs to implement the steps in both of the flow charts given in Figures 5.6 and 5.7.

With some hardware modifications, the system described above will be able to tolerate a failure of one stage of the pipeline. The design of such a fault tolerant pipeline processor is described in the next chapter.

Figure 5.7 Flow of operations in the First Stage

Chapter Six

A Fault Tolerant Pipeline Processor

In this chapter, the design of a Fault Tolerant Pipeline Processor is described whose operation is not affected by the complete failure of a pipeline stage. The design is based on Transputers [63]. Besides fault tolerance, the system has all the processing capabilities of the Adaptable Pipeline described in the previous chapter. The Fault Tolerant Pipeline Processor is extremely versatile as well as reliable and can be used in mission critical applications.

The Fault Tolerant Pipeline Processor has the following architectural features:

- each pipeline stage is capable of executing any operation.
- in case of a fault, a pipeline stage generates an error signal, and the faulty stage is purged out of the system. Computation continues in the pipeline with graceful degradation of performance caused by the complete loss of a pipeline stage.
- the system forms pipelines with a variable number of stages. This means that each computational process activates the number of stages in the pipeline which matches the number of consecutive operations it implements i.e., an instruction implementing w operations will pass through w stages and complete its execution.
- the system provides for fast exchanges of temporary results between pipeline stages.
- the system adapts to varying number of parallel program streams by partitioning resources to form a variable number of parallel pipelines. For example, a system with ten stages may reconfigure to form one pipeline with six stages and two pipelines with two stages each depending on the number of parallel program streams.

For achieving fault tolerance in a pipeline processor, it is essential that the system be capable of bypassing a faulty stage. In a conventional pipeline system, each stage of the pipeline is dedicated to perform a particular type of operation making the functional substitution of a faulty stage with a fault free one impossible. This problem can be overcome by designing a pipeline system based on the universality of a pipeline stage i.e., any stage is capable of executing any operation. Continuation of operation becomes possible if the functions of the faulty stage are carried out by the fault free stage next in line. This requirement is satisfied if each stage of a pipeline processor is constructed from a Transputer. Use of Transputers in the design of the Fault Tolerant Pipeline Processor has another advantage. A Transputer can generate an error signal in case of malfunction and this signal can be transmitted to the external world through an Error pin [30].

Each Transputer can perform self testing by executing a program which tests all its functional blocks. In case of malfunction, it generates an error signal, which is then used by the system to bypass the faulty stage and the functions of the faulty stage are carried out by the fault free stage adjacent to it. Although the Adaptable Pipeline and the Fault Tolerant Pipeline Processor have basically the same design principles, the latter has additional hardware for incorporating the fault tolerance capability. Some of the design features are exactly the same in both of these systems. For example, instructions in both the Adaptable Pipeline and the Fault Tolerant Pipeline Processor have the format discussed in section 5.2 of the previous chapter. However, to facilitate understanding the operation of the Fault Tolerant Pipeline Processor, all the design aspects are described completely in this chapter, even though some of these may seem repetitive. Here the prime emphasis is on the fault tolerant capability of the proposed system which is illustrated when the Fault Tolerant Pipeline Processor is used for

executing a complex arithmetic operation.

6.1 Hardware Organisation of the Fault Tolerant Pipeline Processor

Each stage of the Fault Tolerant Pipeline Processor is built from a Transputer, whose architectural features are given in [63]. Figure 6.1 shows a Transputer and the links which will be frequently referred to in the present discussion.

Figure 6.1 Links of a Transputer

A linear array of s , 32 bit Transputers (the T424 chips [30]) forms the

consecutive stages of the Fault Tolerant Pipeline Processor, shown in Figure 6.2. The stages are named $T_0, T_1, T_2, \dots, T_{s-1}$ and have addresses $0, 1, 2, \dots, s-1$. The Transputer array is connected to a computer Co , which acts as a supervisor. Computer Co stores instructions and data in its memory Mo and fetches them to the first stage of a pipeline. An instruction is initiated for execution from the first stage of the pipeline. After being processed by the first stage, the results and the instruction are sent to the second stage. The second stage then executes the operation assigned to it and passes the refined result and the instruction to the third stage for further processing. The process continues till the instruction is completely executed. After an instruction has been completely executed, the results are sent back to Co for storage in Mo .

The instruction sent by Co for processing in the Fault Tolerant Pipeline Processor has the format given in the next section. It has three fields. The operation performed by a particular stage of the pipeline is determined by two of these fields D and w . Like the Adaptable Pipeline, an instruction as it reaches a pipeline stage is interpreted by a set of programs and control is transferred to the appropriate routine on the basis of the contents of these fields. This routine has all the necessary statements for performing the desired arithmetic operation. A computational process partitioned into n subprocesses passes through $n+1$ consecutive stages of the pipeline: the first n stages perform the necessary arithmetic operations while the result is sent back to Co by the $n+1$ th stage.

The link $Lin0$ of all the Transputers is connected to Co . This ensures that any fault free Transputer may receive instructions from Co thereby becoming the first stage of a pipeline.

After the completion of the execution of an instruction, the final result is sent back to Co for it to be stored in Mo . An instruction implements a sequence of w

operations and ends its execution in stage T_w , where the value of w varies with different instructions. That is, any pipeline stage can be the end stage of some instruction. Therefore, every pipeline stage must be capable of sending computational results to the main memory through Co. For this purpose, the link Lout0 of all the Transputers forming the different stages is connected to the supervisory computer Co. Besides dispatching instructions and receiving the results for storage into memory, Co has the task of monitoring the operation of the entire system. All these objectives are satisfied by connecting the links Lin0, Lout0 of the Transputers to Co.

One important feature of the Transputer used in this design is its ability to generate a signal in its Error pin [30], in case of malfunction. It is assumed that a Transputer forming a stage is equipped with a program which can test all the functional blocks inside it and an Error signal is generated if it is found that the Transputer is malfunctioning. But, a Transputer may generate an Error signal even in case of programming errors such as arithmetic overflow. However, in this system, the Transputers are not used for executing application programs but as stages of the Pipeline and only execute pre-tested software routines. Therefore, whenever an Error signal is generated from a Transputer it indicates that the particular stage is malfunctioning. This Error signal is then used for bypassing the faulty stage and for informing the supervisory computer Co about the fault. In case of malfunction, the supervisory computer is informed of it by means of an interrupt. This is achieved by connecting the Error line to the Error Interrupt Block. When Co receives an interrupt, it searches the Transputer array for the malfunctioning stage which had sent the Error signal. Co does not initiate any instruction from that faulty stage.

For each Transputer T_i , the links Lin1 and Lout1 are used to connect it to its successive pipeline stage through Right Transfer Block 1 (RTB 1) and Left Transfer

Block 1 (LTB 1). They are used for right transfer of instruction and data from T_i to T_{i+1} . For the i th Transputer, instruction and data shifted from T_{i-1} are received in T_i in line Lin3, while acknowledge signal is sent in line Lout3 which enters the sending Transputer in line Lin1. The most important feature of the Fault Tolerant Pipeline Processor is its ability to bypass the faulty stage. That is, if stage T_i is faulty, instruction and data from stage T_{i-1} goes to T_{i+1} without entering the Transputer at stage T_i . This is achieved by the logic in RTB 1. As shown in Figure 6.3(a), the RTB1 enables transfer from T_{i-1} to T_i only if Error Line is low. In case it is high, the path from T_{i-1} to T_{i+1} is enabled. The RTB1 has its inputs coming from Lout1. The same logic is followed in Left Transfer Block 1 (LTB1) which passes the acknowledge signal coming from Lout3 of T_{i+1} to Lin1 of T_{i-1} , if stage T_i is faulty. The logic inside LTB1 is shown in Figure 6.3(b).

As each stage of the pipeline needs to have the ability to send temporary results to other stages, one pair of links Lin2, Lout2 go to the Routing Network through which the results are sent to the appropriate stage. If the result of an operation in stage T_i is a future temporary result for another stage T_j , then it is sent through the Routing Network to that stage and written into its local memory. A Transputer T_i intending to send a temporary result to T_j sends a code d_j to the Routing Network. The code activates the appropriate switches to establish the desired connection between the sending Transputer (T_i) and the addressed Transputer (T_j) through the serial links. If s is the total number of stages in the Fault Tolerant Pipeline Processor, then the

(a) Logic inside Right Transfer Block 1 (RTB 1)

(b) Logic inside Left Transfer Block 1 (LTB 1)

(c) Logic inside Right Transfer Block 2 (RTB 2)

Figure 6.3 Logic inside the Transfer Blocks

number of bits required in d_j for encoding the addresses of all the stages is $\log_2 s$. After establishment of the connection, the bits constituting the code d_j are allowed to proceed to the addressed Transputer T_j through its serial links and an acknowledge signal is sent back from it to the sending Transputer (T_i). After the acknowledge signal is received, T_i can start sending the data value of the temporary result.

In such a pipeline, a stage T_i may send some data to stage T_j , which may be faulty. In the Fault Tolerant Pipeline Processor any stage is capable of performing any operation, so if stage T_j is faulty then all its functions are carried out by stage T_{j+1} . Therefore, unimpaired performance can be achieved by routing all the data coming into stage T_j to stage T_{j+1} . But some other stages T_k ($k \neq j$) in the pipeline may be transferring data to Transputer T_{j+1} and unless this data is shifted to T_{j+2} , stage T_{j+1} will be overburdened and the advantage of pipelining will be lost. Similarly, it is necessary to shift the data coming to stage T_{j+2} to T_{j+3} and so on. This lateral shifting of data from stage k to stage $k+1$ is achieved by means of the logic contained in the block named Right Transfer Block 2 (RTB 2). The input to this logic block in position i (shown in Figure 6.3(c)) come from Error lines of stage 0 to stage i . When any of these Error lines is high, there is a lateral shift of the data i.e., data coming to stage k through the Routing Network is shifted to stage $k+1$.

With the logic proposed here, the Fault Tolerant Pipeline Processor can continue performing computations when a single stage is faulty.

Each pipeline stage T_i uses its 4K local memory for storing data and the

programs necessary for the interpretation of the instructions sent to it either by Co or the previous stage in the pipeline. It also stores temporary results required by processor in Transputer T_i . Each Transputer has an external memory connected to it. This is used when the Transputer's local memory is not sufficient to hold all the programs and data required for the operation of that stage.

6.2 Format of an Instruction

The instruction sent from Co for execution in the Fault Tolerant Pipeline Processor has the format shown in Figure 6.4 and has three fields. In order to specify the operation, it is necessary that each instruction contain an operation code (D). However, when the same D propagates through the pipeline stages, it will activate the same operation in each stage. Selective activation of an operation assigned to stage T_i by an instruction is achieved with the help of another field w.

The field w gives the number of consecutive pipeline stages necessary for the execution of that instruction. Its initial value equals the number of consecutive

Figure 6.4 Instruction Format

operations required for the realisation of that instruction. When an instruction propagates through a stage of the pipeline, the value of w is decremented by 1 and the instruction is passed on to the next stage with this modified value of w. When an instruction reaches a pipeline stage for execution, the contents of the field w give the

number of stages remaining to be traversed before completion of execution of that instruction. A value 0 in the w field implies that the instruction has been completely executed.

The size of the field w is $\log_2 F$ bits, where F is the maximum number of stages of the pipeline required for the execution of any computational process. For executing computational processes which do not require looping, i.e., when the results of a stage do not have to be fed back as inputs to a previous stage, the value of F can be taken to be equal to the maximal length of the pipeline.

The operation performed by a stage is determined by the values of D and w . The instruction reaching a stage is interpreted by a program contained in that stage and perform the operation specified by the combination of D and w . The program to which control is transferred for performing the necessary arithmetic operation is determined by the contents of the fields D and w . The instruction after interpretation by a stage i is passed on to the next stage $i+1$, which performs another arithmetic operation on the results obtained in the previous stage. An instruction passes from one consecutive stage to another till the value of w becomes 0. When any stage i of the pipeline, receives an instruction with 0 in the w field, it immediately knows that the instruction has been completely executed and the results have to be sent back to Co .

In addition to the adaptation codes D and w , each instruction has a field containing the Process Identification Number (PIN). Each computational process (for example an instruction for the execution of a complex arithmetic operation) to be executed in the pipeline is assigned a Process Identification Number by the supervisory computer Co , when it is sent to the first stage of the pipeline. Since a pipeline with s stages can have atmost s different instructions residing in its different stages, the values of PIN range from 0 to $s-1$ and the number of bits b required in the PIN field has to

satisfy the relation $b = \log_2 s$. For example, a 16 stage pipeline can have PIN values from 0 to 15. Therefore, the instruction format for such a pipeline needs to have 4 bits in the PIN field. Whenever an instruction is dispatched by Co for processing in the Fault Tolerant Pipeline, a number between 0 and $s-1$ is assigned to it in this field such that this number is different from the PIN values of all the instructions currently residing in the pipeline. The contents of the PIN field serve a number of purposes. For example, the data value which is sent from one stage to another is always preceded by the value of PIN to identify the computational process to which it belongs while the program which interprets the operation code of an instruction is referred to by the value of D and w. Since an instruction has fields containing both PIN and D values, the correspondence between the interpretive program and the variables can be established easily.

6.3 Operation of the Fault Tolerant Pipeline Processor

It has already been stated, that a Fault Tolerant Pipeline Processor designed with Transputers has all the processing capabilities of the Adaptable Pipeline described in the previous chapter. In this section, its mode of operation is illustrated when applied to the high speed execution of a complex arithmetic operation (say an instruction for floating point arithmetic) partitioned into suboperations for execution in different stages of the pipeline. The Transputer of each stage is pre-loaded with all the programs necessary for the execution of the suboperations of different computational processes that may be executed in the pipeline. Each stage selects the particular program needed for performing a suboperation of an instruction on the basis of the contents of the D and w fields of the incoming instruction. When a stage becomes faulty, a fault free stage has to execute the suboperation which is normally carried out by the stage adjacent to it. This

is because, when T_i becomes faulty, instruction and data are shifted from T_{i-1} to T_{i+1} , by means of the hardware described in section 6.1. The functions that are normally carried out by T_{i+1} are now carried out by T_{i+2} . T_{i+3} now performs the functions of T_{i+2} and so on. That is when a stage T_i becomes faulty, all stages after T_i i.e., T_k ($k > i$) has to perform the functions of T_{k-1} . As any stage of the pipeline processor may become faulty, the suboperations to be performed by a particular stage can not be determined *a priori* as in the case of the Adaptable Pipeline, but a stage needs to be equipped with all the programs needed for performing all the suboperations. Accommodating all this software is not a problem as a Transputer can directly access 4GB of external memory.

- The operation of the system is now illustrated by assuming that the Fault Tolerant Pipeline Processor forms a single pipeline with 16 stages and the stages of the pipeline are T_0 to T_{15} , having addresses 0, 1, 2, ..., 15. The transputers forming the stages are interconnected through the serial links and the logic blocks as discussed in section 6.1.

Co, the supervisory computer sends instructions to the first stage of the pipeline. Since it is assumed that the system forms only one pipeline, the first stage is at T_0 . However, if T_0 is faulty, Co does not send instructions to it but initiates execution of instructions from T_1 . The instructions sent to T_0 have the format given in section 6.2. After sending the instruction, the Supervisory Computer Co sends the operands required for the execution of this instruction. This is done in the same way as sending the instruction except that before sending the bit values of the operands, the value of the PIN is also sent to T_0 . This protocol enables the Transputer to identify the

computational process to which the operands belong. The operands received from Co are stored in the workspace of the process whose name corresponds to the value in PIN. In the terminology of Transputers, the workspace denotes an area of on-chip memory where local variables are kept. After T_0 receives the instruction and data, the instruction is interpreted to determine the type of suboperation the stage has to perform and control is transferred to the appropriate routine. These routines are stored as processes (in the terminology of OCCAM) with their names corresponding to values in the D and w fields. When control is transferred to a process for performing the assigned suboperation, the variables needed by this process are fetched from the workspace of the process whose name corresponds to the value in the PIN field of the instruction.

When the arithmetic operation assigned to a stage has been performed, the value of w is decremented by 1. The instruction with this modified value of w is then sent to the next stage for further processing together with the result(s) of the arithmetic operation. The process continues till the value of w becomes 0.

When a stage receives an instruction with a value 0 in its w field, it immediately recognises that the instruction has been completely executed and its results have to be sent to Co. When a stage intends to transmit a result to the supervisory computer Co, it first sends the value of the PIN on the serial link and then the bits corresponding to the result are sent. This protocol enables the supervisory computer to establish a correspondence between the instruction sent for processing on the Fault Tolerant Pipeline Processor and the result(s) obtained after the completion of processing. If after the end of some suboperation, a stage i intends to send some temporary result(s) to another stage j, where $j \neq i+1$, the Transputer in stage i first sends the identification code d_j of that stage to the Routing Network. The Routing Network establishes the connection between the serial links corresponding to i and j and data

transmission from T_i to T_j takes place. Before sending the actual value of the data, the value of PIN is also transmitted to identify the computational process to which it belongs. However, if T_j is faulty, the data together with the PIN value does not enter T_j but is shifted to T_{j+1} as discussed in section 6.1, without T_i knowing about it. The arithmetic operation which was to have been done by T_j is now done by T_{j+1} .

From the above discussion, it is clear that the operation of the system is not affected by failure of a stage of the pipeline as the function of the faulty stage can be carried out by a fault free stage allowing the execution of an instruction to continue unhampered.

6.4 Organisation of a Pipeline Stage

Like the Adaptable Pipeline the whole system needs to be programmed in OCCAM language [31] to gain the most from the use of Transputers. The software organisation of the Fault Tolerant Pipeline Processor is same as that of the Adaptable Pipeline. The reason behind using the same software organisation is that, it enables the operation of the system to continue without any modification to the routines even when a stage of the pipeline fails. The organisation of the software is described below in terms of the OCCAM model of processes and channels.

The interconnection of the links between the successive Transputers has been discussed in section 6.1. Each of these unidirectional links implements an OCCAM channel. Each Transputer has six channels which go in the directions shown in Figure 6.5.

Figure 6.5 Channel Organisation

The links Lin0, Lout0 constitutes Channel 0 (Ch0) and Channel 1 (Ch1) respectively. Link Lout1 form Channel 2 (Ch2). Instructions and data enter T_i from preceding stage through the link Lin3 constituting Channel 5 (Ch5). After the execution of an arithmetic operation, a pipeline stage may need to send the results of the computation not only to its next stage but to another stage T_j through the link Eout2 which forms Channel 4 (Ch4). Data enters the Transputer T_i from other stages via the Routing Network through link Lin2, which constitutes the Channel 3 (Ch3).

The flow chart in Figure 6.6 gives the sequence of operations followed by a stage. The transputer in each stage of the pipeline can get inputs from two channels: Ch5 and Ch3. Each transputer should be programmed to be able to input from these two channels concurrently. As different pipeline stages require different amounts of time for processing, for each stage the time of arrival of data in Ch3 is independent of the time of arrival of instruction and data in Ch5.

2 of/de 2

MicroD

Figure 6.6 Flow of operations in a pipeline stage

An instruction as it reaches a stage is analysed and control is transferred to a routine depending on the values in the fields D and w in the incoming instruction. Besides statements for executing the assigned suboperation, this routine contains processes for sending the instruction and data to other stages. The statements in the output process have to take into account the protocol required to be followed for the purpose. For example, if the Transputer in stage T_i intends to send a data value to another Transputer T_j through the Routing Network, then the output process must contain statements for first outputting the value d_j . This is followed by statements for transmitting the PIN value of the instruction to which the data belongs. Finally, T_i transmits the bits giving the value of the data. Although the address d_j is specified when transmitting data to stage T_j , if T_j is faulty, external hardware automatically shifts the data from stage T_j to stage T_{j+1} without any need for address modification.

As previously discussed, when a fault occurs there is a necessity for shifting the functions of a stage to its adjacent one. The main advantage of the software organisation described above is that no modification to the routines is necessary for this purpose. Therefore, in case of a fault operation of the pipeline continues unhampered.

The Fault Tolerant Pipeline Processor whose design and operation has been described above can withstand the failure of a single stage. However, more complicated circuitry have to be used for tolerating multiple stage failures.

Chapter Seven

A Nomenclature for Parallel Processing Systems

In the previous chapters, the various issues involved in the design of pipeline processing systems have been discussed. While the technique of pipelining has been widely used in today's supercomputing systems, there are parallel processing systems which utilise other techniques to reduce the execution time of an algorithm. Today, parallel processing systems include algorithmically specialised VLSI processors and computers with a single CPU capable of executing several instructions or computing several distinct data items simultaneously, as well as multiple CPU systems where different CPUs work simultaneously on independent tasks of the same program.

The parallel processing systems that have evolved are usually referred to in the literature by terms such as Systolic Arrays, Array Processor, Associative Processor, Pipeline Processor, Multiprocessor, Multifunction Processor, Data Flow Processor, Demand Driven Processor. Unfortunately however, these terms do not give precise information about the dominant characteristic features or the capabilities of these systems. For example, a computer system consisting of a number of CPUs is termed a multiprocessor or a multicomputer system even though it may have any one of the strongly contrasting control patterns as centralised control, distributed control, data flow control. Moreover, although a few of the most recently proposed designs fall precisely into the aforementioned types, most current designs are hybrids which combine the concepts from two or more of these. Today, because of the low cost of hardware, such hybridation has become a rule rather than exception and there is no way in which the dominant characteristics of such hybrid systems can be described by the use of simple

terms. There is a need for a taxonomy which can bring out the diversity in the designs of parallel processing systems as well as enable the specification of their salient features in simple terms. The major problem in formulating such a taxonomy is choosing the minimum amount of detail that should be incorporated in the classification scheme so as to make the nomenclature complete as well as compact. A taxonomy which meets these two conflicting requirements will permit very compressed descriptions of the system. It will employ only those features of the architecture that are of interest, ignoring the rest. There have been some attempts to develop such a taxonomy, but all these have been found to be inadequate.

In this chapter, a nomenclature of parallel processing systems is proposed based on system characteristics which are germane to these architectures. The nomenclature is based on the classification scheme proposed in [6.1].

7.1 Inadequacies of Existing Taxonomies

Flynn in 1966 initiated the classification of high speed computing systems on the basis of multiplicity of hardware required to service the instruction and data streams [16]. He divided computer systems into four classes: confluent Single Instruction stream Single Data stream machine (SISD), Single Instruction stream Multiple Data stream machine (SIMD), Multiple Instruction stream Single Data stream machine (MISD), Multiple Instruction stream Multiple Data stream machine (MIMD). According to this classification, a uniprocessor is an SISD computer, a multiprocessor is an MIMD computer and an Array Processor is an SIMD computer. No system exists which can be termed MISD. Unfortunately however, some architectures can be placed in either of two classes depending upon one's concept of instruction and data stream. For

example, Flynn originally considered Pipeline Processors to be confluent SISD [16]; later he considered Pipeline Processors to be of SIMD type [17]. However, other taxonomists (Handler [20], Higbie[23], and Thurber [59]) felt that Pipeline Processors were actually MISD machines. Another drawback of Flynn's classification is that, it has very little discriminatory power. For example, it refers to all systems with multiple processors as belonging to MIMD class irrespective of the characteristics of the individual processors and this class can be stretched to include tightly coupled multiprocessors with sequential control flow, data flow multiprocessors and even distributed processors. Despite these limitations of Flynn's taxonomy the nomenclature suggested by Flynn twenty years back is still widely used for want of an alternative unambiguous and compact naming scheme.

Kuck's classification [41] is similar to Flynn's except that he introduced the ideas of scalar and array instruction stream and scalar and array data stream or execution stream (as Kuck refers to) into the taxonomy. He has only been able to enhance the discriminatory power of Flynn's classification to a small extent.

In 1972, Feng [45] proposed a classification of computer systems according to the serial or parallel mode of processing words as well as the bits contained in that word. His classification is more useful for differentiating between the various types of Array and Associative Processors than for grasping the important design aspects of the various high speed processing architectures.

Baer in his paper [5] has discussed several attributes for classifying a computer system starting with functionality i.e., whether the system is general purpose or special purpose. Giloi [18] suggested a taxonomy based on the hardware structure as well as on the operational principle i.e., the information representation in the machine. Both Baer and Giloi have given an exhaustive list of attributes for classifying a

computer system. Their interest in discussing these taxonomies has been for teaching computer architecture principles rather than for developing a nomenclature for system identification.

The rapid decrease in the cost of hardware has resulted in a spurt in the growth of nonconventional systems consisting of a large number of processors. In order to distinguish between them and avoid the use of ubiquitous terms there is a need to include some characteristic feature in the taxonomy which can bring out the difference in various system architectures. This is what is done here. A classification is done on the basis of qualitative characteristics which it is felt are the dominant features of a computer system in order to develop a nomenclature for these systems. Quantitative measures such as the number of control units, the number of ALUs controlled by each unit etc. (as suggested by Handler [21] for describing a system) are avoided.

7.2 Design Decisions and the Taxonomical Tree

A parallel processing system is the result of a series of design decisions and as shown in Figure 7.1, the design space can be considered to be a tree. The root of the tree is a parallel processing system which attempt to reduce the execution time of an algorithm. Below this, are levels representing choice of the level of concurrency, method of algorithm realisation, the execution characteristic and control structure all of which contribute to the identification of a system. Some spaces in the tree are vacant as these features might be of no interest for a particular design.

The first strategic choice is the *level of concurrency* that is to be exploited in the parallel processing system. The same operation may be performed on multiple data items simultaneously. This is referred to as *data level concurrency*

Figure 7.1 The Taxonomical Tree

(represented by **D** in the naming scheme) as in the case of vectors where each element of the vector undergoes transformation together with the transformation of other elements of the vector. Ability to perform more than one arithmetic logical operation simultaneously as specified in successive instructions in a program is referred to as *operation level concurrency* (represented by **O** in the naming scheme). Instructions which are independent can proceed for simultaneous execution. Besides these two, a system may be designed with necessary software support to exploit concurrency at the level of task which is composed of a block of instructions. Processes or tasks represent activities that may themselves be sequential but can be executed concurrently and asynchronously with other tasks in a system. Some programming languages like PL/1, ADA, Algol 68 have incorporated features to explicitly specify tasks which qualify for simultaneous execution and for the synchronisation of concurrent processes. A number of CPUs interconnected to form a multiprocessor system or a distributed processing system can exploit this *task level of concurrency* (represented by **T** in the naming scheme). Each of these tasks is run on a processor which may exploit still another level of parallelism and may have the hardware support for exploiting operation level concurrency for high speed execution of individual tasks. In this case, the multiple processor system composed of independent processors is described by the characteristics of individual processors. Accordingly, the terminology of the composite system contain terms used for referring to the architectural features of individual processors constituting the system.

The second level of the taxonomical tree considers methods of *algorithm realisation*. With the advent of VLSI, it is possible to realise some complex algorithms each in a single VLSI chip. This level distinguishes parallel processing systems which realise an algorithm by *VLSI circuit implementation* (represented by **C** in the naming

scheme) from structures in which algorithms are realised by *program execution* (represented by **P** in the naming scheme). The algorithm which can be circuit implemented will depend on the function of each processor and interconnection structure of the processing elements integrated in the chip. It has been shown by Kung [45] that algorithms for matrix multiplication, FFT, LU decomposition etc. can be easily and efficiently realised by such VLSI chips. The stream of data is acted upon by the processing elements and data items move from one processing element to another in a predetermined way in such a system. These special purpose devices can be considered for the hardware replacement of many of the subroutines in a scientific software library. Alternatively, a computer system may realise an algorithm by executing a sequence of instructions stored in the memory of a computer system, which has been the conventional way ever since the evolution of computers.

The next level in the taxonomical tree considers the *execution characteristic*. In systems realising an algorithm by program execution, the operation of instruction processing may be broken into suboperations to be performed in different stages working simultaneously, each stage performing a specific suboperation and the result of one stage going into the next. This method of achieving high speed by overlapping the suboperations of successive computations is referred to as *pipelining* (represented by **Pi** in the naming scheme) and can be used to realise an algorithm whenever that algorithm can be divided into a fixed number of steps that are to be executed in sequence. For non-pipelined systems high speed computation may be achieved by simultaneous or *parallel operation of processing elements* (represented by **Pa** in the naming scheme). Multiple arithmetic units may be activated simultaneously. For example, in non-pipelined architectures exploiting data level concurrency by program execution, one instruction may operate simultaneously on a number of data

items. In circuit implemented systems, the processing elements may be working simultaneously on the data stream. In systems exploiting operation level concurrency, independent instructions may be sent to the multiple arithmetic units capable of simultaneous operation. An Instruction Pipeline may have multiple arithmetic units capable of performing arithmetic operations simultaneously. However, they still belong to the category of pipeline processor as instruction pipelining is the dominant feature in their design.

The fourth level is concerned with the choice of the method of *system control*. A system realising an algorithm by program execution may employ synchronous or asynchronous method for instruction sequencing. In conventional computer architectures, instructions are selected one after another and program counter incremented. This is referred to as *synchronous control* (represented by S in the naming scheme) in the sense that fetching of instructions is in synchronism with the contents of the program counter. In contrast to this, is the sequencing method followed in Data Driven and Demand Driven architectures [61]. In Data Driven architecture, instructions are sent for execution when its operands are ready. In this case the flow of control is the same as the flow of data and there is no concept of program counter. In a computer having Demand Driven architecture, instructions are selected when the value they produce is needed by another already selected instruction. Data Driven and Demand Driven computer systems show the maximum deviation from Von Neumann architecture and these can be considered as having *asynchronous control structure* (represented by A in the naming scheme). On the other hand in systems realising an algorithm by circuit implementation, operands may move in the VLSI chip synchronously or asynchronously. In this case, the term synchronous/asynchronous is used to denote whether not the processing is carried out in accordance with external timing signals.

The final level of the taxonomical tree comprises the leaf nodes representing specific system designs. As shown in the taxonomical tree (Figure 7.1), the nomenclature of different systems is composed of terms used to express their capabilities in the four lower levels. For example, *D* denotes the capability of a system to exploit *data level concurrency*, *P* specifies that the system realises an algorithm by *program execution*, *Pa* denotes that the system realises an algorithm by *parallel operation of PEs* and *S* denotes that it has a *synchronous control structure*. A system with all these capabilities is referred to by **DPPaS**.

7.3 System Architectures and the Proposed Taxonomy

A study of parallel processing systems that have evolved reveals that these can generally be divided into Systolic Array, Pipeline Processor (Instruction Pipeline), Multiprocessor, Multifunction Processor, Associative processor, Array Processor, Data Flow Processor. Besides these, research is being carried out on Demand Driven computer architecture. Haynes et. al. in [22] have discussed the salient features of these machines. In this discussion, the aforementioned types of parallel processing systems is referred to as 'pure' parallel processors in order to differentiate them from hybrid systems which combine the features of these 'pure' parallel processing systems.

Without going into the details of implementation of these architectures, it is shown how the important features of these 'pure' parallel processing systems can be described by the terms suggested in the leaf nodes of the taxonomical tree. Further, it is shown how the processing characteristics of the hybrid systems can be described by a combination of terms used for referring to the features of the 'pure' parallel processing systems.

With the advent of VLSI, intensive research is being carried out on how best to utilise this integrated circuit technology. Of the various architectures that have been proposed to take advantage of Very Large Scale Integration, Kung's Systolic Arrays [44] seems to be well suited as special purpose devices for high speed implementation of complex algorithms (termed Systolic Algorithms), which have been traditionally realised by writing large programs. Study of Systolic Systems having communication geometry of 1-dimensional linear array, 2-dimensional square array, 2-dimensional hexagonal array suggests that the terms **DCPaS**, **DCPaA** can be used to describe such systolic systems depending on whether data flows through the system synchronously or asynchronously. Systolic Trees have been proposed by H.T. Kung [44] for the evaluation of arithmetic expressions. **OCPaS** or **OCPaA** may be used for referring to such systolic systems, which operate on scalar data items.

— In this discussion, the term pipeline processor is used for Instruction Pipeline. These systems exploit instruction level concurrency and they can be referred to by **OPPiS**.

Pipelining finds its most natural application in vector processing. If the same operation is performed on all the elements of independent vectors e.g., adding two vectors then a continuous excitation of a pipeline can be obtained. Hence the term **DPPiS** can be used for such pipelines. Again some pipeline processors have vector instructions in their instruction sets with additional hardware for routing the operands between the pipeline segments. They have been termed in the literature as Vector Processors [52]. If a Vector Processor is considered as a pipelined processor exploiting data level concurrency then it can be referred to by **DPPiS**.

Multiple arithmetic units operating in lock step and performing the same operation as specified in an instruction on different data items form an Array Processor

[26]. Both Array and Associative Processor [65] belong to **DPPaS** category but differ in the organisation of the attached memory.

Multifunction Processor [41] with multiple arithmetic/logic units under a centralised control and exploiting operation level concurrency as in CDC 6600 [58] may be described by the term **OPPaS**.

In [14] Dennis has discussed a type of instruction execution mechanism followed in some Data Flow Processors. An instruction is fetched from memory as soon as it is ready for execution, i.e., all its operands are available and it passes through sequential stages of processing. This has been called a Circular Pipeline and the term **OPPiA** can be used for such data flow processors. Study of systems with data driven architecture reveals that some other data flow machines have been built with the token matching mechanism [61] and they may be described by **OPPaA**.

Multiple CPU systems may exploit concurrency at the task level depending on the system software support. If the constituent processors do not exploit data or operation level concurrency, then only the term **T** is used for these systems. If however the individual processors exploit instruction level concurrency for high speed execution of each independent task i.e., if there are two levels of parallel processing, then the system is described by **T** and the appropriate term suggested under operation level concurrency with an intervening '*' sign. The '*' sign is used for parallel processing systems capable of exploiting more than one level of parallelism. This will be discussed further.

As already stated, if a computer system has hardware support to give it a processing capability that is a combination of the features found in parallel processing systems exploiting data/operation level concurrency then the system is described by a combination of terms given in the leaf nodes of the taxonomical tree. For example, most

pipeline processors have hardware for vector processing in addition to scalar processing facility. The capability of such a system can be described by an additive combination of terms used for describing each of these features. Therefore the characteristics of such a system is described by $OPPiS + DPPiS$. Examples are T₁ ASC and CDC Star-100. Again some systems may be equipped with hardware to allow concurrent processing of vector instructions which do not interfere with each other, as in CRAY-1 [55]. Such a facility may be considered to be exploiting parallelism at two levels - first between vector instructions and second at the data level in processing a vector instruction and this may be described by $O * DPPiS$. The first 'O' with the '*' sign denotes concurrency between vector instructions and the capability of the system to exploit more than one level of parallelism for reducing a program's execution time. As CRAY-1 has facilities for scalar processing in addition to facility for concurrent processing of vector instructions, the term $O * DPPiS + OPPiS$ can be used to describe CRAY-1.

Illustrating further the application of the proposed nomenclature in describing the multifarious capabilities of the different parallel processing systems, a multiple CPU system is considered where each CPU is itself a pipeline processor. If such a system has software to support concurrent execution of tasks on different CPUs and each CPU employs instruction pipelining for high speed processing of individual tasks, then evidently the system exploits parallelism at two levels for reducing the execution time of a program. Hence such a system can be described by $T * OPPiS$. An example of a real system which utilises this concept is S-1 [64].

Proceeding in a similar way, the salient features of CRAY X-MP system [7] which has two CRAY-1 like processors and which employs multitasking for running related tasks of a single job on two processors can be described by $T * (O * DPPiS + OPPiS)$.

Numerous other examples can be given to show the application of the nomenclature developed in this taxonomy for specifying the predominant architectural features of parallel processing systems that have been built. It is felt that the nomenclature proposed here is more meaningful in describing parallel processing systems at the macroscopic level than by a specification of multiplicity of instruction and data streams as per Flynn's classification, which is still the most widely used taxonomy.

Chapter Eight

Concluding Remarks

The concern of this thesis has been with problems occurring in pipeline processing systems. Both Instruction Pipeline and Arithmetic Pipeline have been investigated and solutions have been presented for some of the fundamental problems occurring in these systems.

An Instruction Pipeline achieves maximum throughput when successive instructions are independent but there is a disruption of flow when a branch instruction is encountered. Most of the solutions to the branch problem attempt to predict whether or not a branch will be taken. If the prediction is correct, then the initiation of the correct sequence of instructions continues without delay. The performance of an Instruction Pipeline depends on the prediction accuracy i.e., on the probability of correct branch prediction. In this thesis, the performance of an Instruction Pipeline has been analysed to give an expression for efficiency or utilisation of pipeline segments relating it to the prediction accuracy. The expression for efficiency deduced here from the space-time or the geometric model takes into consideration that generally instruction sets of computers contain two types of conditional branch instructions: those which branch to a target address depending on the testing of the condition code set by some previous instruction and those which transfer control to some address depending on the outcome of certain computation which it performs. To the author's knowledge, this the first time that such an expression has been given. It enables one to study the sensitivity of the performance of an Instruction Pipeline to the number of conditional branch instructions and the probability of correct branch prediction. Although it is extremely difficult to obtain the value of 1 for prediction accuracy, this study reveals that a value of 0.8 can give a

reasonably good value of utilisation of pipeline segments. The simplicity of the analysis followed for obtaining the values of the necessary parameters makes the methodology particularly attractive.

The technique of pipelining has been widely used in reducing the execution time of complex arithmetic operations like the algorithms for floating point arithmetic. Conventional Arithmetic Pipelines suffer from the problem of inflexibility and tends to be dedicated to a particular computational process. This drawback can be overcome in an Adaptable Pipeline with the architectural features outlined in this thesis. The design of an Adaptable Pipeline has been proposed which can be used for the processing of different types of computational processes concurrently and which can be implemented with commercially available VLSI chips called Transputers. The software organisation of a pipeline stage has been described in terms of OCCAM model of processes and channels, thereby facilitating writing of OCCAM programs for the purpose. As the entire operation of the system is based on software, the proposed system is very flexible. Its area of application can be changed by simply altering the OCCAM programs in the Transputers according to the requirements.

An important problem facing pipeline systems is the disruption of flow of instruction and/or data caused by the failure of a stage. In this thesis, the design of a Fault Tolerant Pipeline Processor has been proposed whose operation is unimpaired even when a stage fails. It has the ability to bypass the faulty stage so that the computation can continue. Like the Adaptable Pipeline, its design has been based on an array of Transputers but it has additional hardware for incorporating the fault tolerance capability. The Fault Tolerant Pipeline Processor has another advantage. It has all the processing capabilities of the Adaptable Pipeline making it extremely versatile. It is felt that the fault tolerant capability and versatility of the Fault Tolerant Pipeline Processor will make it

- useful in many mission critical applications.

Besides Instruction and Arithmetic Pipeline, there are other parallel processing systems which utilise innovative architectures for reducing the execution time of an algorithm. The salient architectural features of these systems can no longer be adequately described by any of the nomenclatures proposed till date. A taxonomy has become essential such that all the systems can be clearly identified. In this thesis, a classification has been proposed on the basis of level of concurrency exploited, method of algorithm realisation, execution characteristic and system control. A nomenclature has been developed for describing the parallel processing computer systems which exploit concurrency at the data, operation or task level. There is no doubt that the terms suggested in the taxonomy proposed in this thesis will be more meaningful in describing all types of parallel processing systems at the macroscopic level than by a specification of multiplicity of instruction and data streams as per Flynn's classification, which is still widely used.

Although solutions to some important problems occurring in pipeline processing systems have been proposed in this thesis, it is felt that further research may be carried out in the following areas:

- i. Developing an *optimal* prediction strategy for an Instruction Pipeline i.e., a strategy which will yield a prediction accuracy of 1.0.
- ii. Designing an Adaptable Pipeline in which each stage is able to modify its word size according to the requirements of different computational processes [2]. This will save computation time by eliminating the need for handling wordlengths greater than the desired precision.

iii. For matching the type of computations encountered in different algorithms, it is desirable to have a system which can assume any of the following architectural types:

- Multiprocessor,
- Array Processor,
- Pipeline Processor,
- mixed, meaning coresidence of any combination of the architectures indicated above.

Such a supercomputing system [38] can speed-up the execution of any algorithm by reconfiguring itself to assume any one of the parallel processing architectures needed to match the type of algorithm which it has to execute. Designing such a system has been the subject of intense research [37]. While the design of an Adaptable Pipeline has been proposed in this thesis, research may be directed at modifying the proposed design so that it can reconfigure to form Multiprocessor and Array Processor depending upon the computational requirements.

iv. In this thesis, the design of a Fault Tolerant Pipeline Processor has been proposed whose operation is unimpaired when a stage fails. While such a system can withstand a single fault, the proposed design is unable to cope with multiple stage failures. Because of the important role being played by pipeline processing systems in mission critical applications, it is felt that research should be directed at designing Pipeline Processors whose operation is not affected by multiple faults.

References

- [1] Agrawal, P. and Agrawal, R., 'Software Implementation of a Recursive Fault Tolerance Algorithm on a Network of Computers', Proc. Thirteenth Annual International Symposium on Computer Architecture, June 1986, pp. 65-72
- [2] N.A. Alexandridis, 'Adaptable Software and Hardware: Problems and Solutions', IEEE Computer, Vol. 19, No. 2, February 1986, pp. 29-39
- [3] Amdahl Corporation, Amdahl 470 V/6 Machine Reference Manual, Sunnyvale, California, 1975
- [4] Anderson, D.W., Sparacio, F.J. and Tomasulo, R.M., 'The IBMSys/360 Model 91: Machine Philosophy and Instruction Handling', IBM Journal of Research and Development, Vol. 11, No. 1, January 1967, pp. 8-24
- [5] Baer, J.L., 'Whither a Taxonomy of Computer Systems', TR-82-10-01, Department of Computer Science, University of Washington, Seattle, 1982
- [6] Baskett, F. and Keller, T.W., 'An Evaluation of the CRAY-1 Computer', in Kuck et.al.(Eds.), High Speed Computer and Algorithm Organisation, New York: Academic Press, 1977, pp. 71-84
- [6.1] Basu, A., 'A Classification of Parallel Processing Systems', Proc. 1984 IEEE International Conference on Computer Design, October 1984, pp. 222-225
- [7] Chen, S.C., 'Large Scale and High Speed Multiprocessor Systems for Scientific Applications: CRAY X-MP Series', Proc. NATO Advanced Research Workshop on High Speed Computing, Springer Verlag, Julich, W.Germany, 1983

- [8] Chen, T. C., 'Parallelism, Pipelining and Computer Efficiency', Computer Design, Vol. 10, No. 1, January 1971, pp. 69-74
- [9] Chen, T.C., 'Overlap and Pipeline Processing' in H.S. Stone (Ed.), Introduction to Computer Architecture, Chicago: Science Research Associates, 1975, pp. 375-431
- [10] Control Data Corporation, STAR 100 Hardware Reference Manual, Arden Hills Minn., 1975
- [11] Control Data Corporation, Control Data 7600 Hardware Reference Manual, Arden Hills, Minn., 1975
- [12] Cray Research Inc., CRAY-1 Computer System Hardware Reference Manual, Bloomington, Minn., 1977
- [13] Davidson, E.S., Thomas, D.P., Shar, L.E. and Patel, J.H. 'Effective Control for Pipelined Computers', Proc. COMPCON '75, IEEE, 1975, pp. 181-184
- [14] Dennis, J.B., 'Data Flow Super Computers', IEEE Computer, Vol. 13, No. 11, November 1980, pp. 48-56
- [15] Feng, T.Y., 'Some Characteristics of Associative/Parallel Processing', Proc. 1972 Segamore Computer Conference, Syracuse University, 1972
- [16] Flynn, M.J., 'Very High Speed Computing Systems', Proc. IEEE, Vol. 54, No. 12, December 1966, pp. 1901-1909
- [17] Flynn, M.J., 'Some Computer Organisations and their Effectiveness', IEEE Trans. on Computers, Vol. C-21, No. 9, September 1972, pp. 948-960
- [18] Giloi, W.K., 'Towards a Taxonomy of Computer Architecture based on the Machine Data Type View', Proc. Tenth Annual International Symposium on Computer Architecture, 1983, pp. 6-15

- [19] Hallin, T.G. and Flynn, M.J., 'Pipelining of Arithmetic Functions', IEEE Trans. on Computers, Vol. C-21, No. 8, August 1972, pp. 880-886
- [20] Handler, W., 'On Classification Schemes for Computer Systems in the Post Von Neumann Era' in Lecture Notes in Computer Science, G.Goos and J. Hartmanis (Eds.), Springer Verlag, New York, 1975
- [21] Handler, W., 'The Impact of Classification Schemes on Computer Architecture', Proc. International Conference on Parallel Processing, August 1977, pp. 7-15
- [22] Haynes, L.S., Lau, R.L., Siewiorek, D.P., Mizell, D.W., 'A Survey of Highly Parallel Computing', IEEE Computer, Vol. 15, No. 1, January 1982, pp. 9-24
- [23] Higbie, L.C., 'Supercomputer Architecture', IEEE Computer, Vol. 6, No. 12, December 1973, pp. 48-58
- [24] Holgate, R.W. and Ibbett, R. N., 'An Analysis of Instruction Fetching Strategies in Pipelined Computers', IEEE Trans. on Computers, Vol. C-29, No. 4, April 1980, pp. 325-329
- [25] Hwang, K., Su, S.P. and Ni, L.M., 'Vector Computer Architecture and Processing Techniques', in Yovits (Ed.), Advances in Computers, Vol. 20, New York: Academic Press; 1981, pp. 115-197
- [26] Hwang, K. and Briggs, F.A., Computer Architecture and Parallel Processing, McGraw Hill Book Company, 1984
- [27] Ibett, R.N., 'The MU5 Instruction Pipeline', The Computer Journal , Vol. 15, No. 1, February 1972, pp. 42-50
- [28] Ibbett, R.N. and Capon, P.C. 'The Development of the MU5 Computer System', Communications of the ACM, Vol. 21, No. 1, January 1978, pp. 13-24

- [29] IBM Corporation, IBM System /360 Principles of Operation, New York, 1970
- [30] INMOS Limited, IMS T424 Transputer Data Sheet
- [31] INMOS Limited, OCCAM Programming Manual, Prentice Hall International, 1984
- [32] Jump, J.R. and Ahuja, S.R., 'Effective Pipelining of Digital Systems', IEEE Trans. on Computers, Vol. C-28, No. 9, September 1978, pp. 855-865
- [33] Kamal, A.K., Singh, H. and Agrawal, D.P., 'A Generalised Pipeline Array', IEEE Trans. on Computers, Vol. C-24, No. 5, May 1974, pp. 533-536
- [34] Kartashev, S.I. and Kartashev, S.P., 'Dynamic Architectures: Problems and Solutions', IEEE Computer, Vol. 11, No. 7, July 1978, pp. 26-40
- [35] Kartashev, S.P. and Kartashev, S.I., 'Adaptable Pipeline System with Dynamic Architecture', Proc. 1979 International Conference on Parallel Processing, August 1979, pp. 222-230
- [36] Kartashev, S.I., Kartashev, S.P. and Ramamoorthy, C.V., 'Adaptation Properties for Dynamic Architectures', Proc. 1979 AFIPS National Computer Conference, Vol. 48, AFIPS Press, pp. 543-556
- [37] Kartashev, S.I. and Kartashev, S.P., Eds. Designing and Programming Modern Computers and Systems, Englewood Cliffs, N.J.: Prentice Hall, 1982
- [38] Kartashev, S. and Kartashev, S., Guest Editors' Introduction, IEEE Computer, Vol. 19, No. 2, February 1986, pp. 9-13
- [39] Kogge, P.M., 'A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations', IEEE Trans. on Computers, Vol. C-22, No. 8, August 1973, pp. 786-793

- [40] Kogge, P.M., The Architecture of Pipelined Computers, McGraw Hill Book Company, 1981
- [41] Kuck, D.J., The Structure of Computers and Computations, Vol. 1, New York: John Wiley, 1978
- [42] Kuck, D.J. and Stokes, R.A., 'The Burroughs Scientific Processor (BSP)', IEEE Trans. on Computers, Vol. C-31, No. 5, May 1982, pp. 363-376
- [43] Kuhl, J.G. and Reddy, S.M., 'Fault-Tolerance Considerations in Large, Multiple Processor Systems', IEEE Computer, Vol. 19, No. 3, March 1986, pp. 56-67
- [44] Kung, H.T., 'Let's design algorithms for VLSI Systems', CMU- CS- 79 -151, Department of Computer Science, Carnegie Mellon University, January 1979
- [45] Kung, H.T., 'Why Systolic Architectures', IEEE Computer, Vol. 15, No. 1, January 1982, pp. 37-46
- [46] Larson, A.G. and Davidson, E.S., 'Cost Effective Design of Special Purpose Processors: A Fast Fourier Transform Case Study', Proc. of Eleventh Allerton Conference, University of Illinois, 1973, pp. 547-557
- [47] Lee, J.K.F. and Smith, A.J., 'Branch Prediction Strategies and Branch Target Buffer Design', IEEE-Computer, Vol. 17, No. 1, January 1984, pp. 6-22
- [48] McFarling, S. and Hennessy, J., 'Reducing the Cost of Branches', Proc. Thirteenth Annual International Symposium on Computer Architecture, June 1986, pp. 396-403
- [49] Patterson, D.A. and Sequin, H., 'RISC-1: A Reduced Instruction Set VLSI Computer', Proc. Eighth Annual International Symposium on Computer Architecture, May 1981, pp. 443-458

- [50] Perrott, R.H., 'A Language for Array and Vector Processors', ACM Trans. on Programming Languages and Systems, Vol. 1, No. 2, Oct. 1979, pp. 177-195
- [51] Ramamoorthy, C.V. and Li, H.F., 'Efficiency in Generalised Pipeline Networks', Proc. AFIPS National Computer Conference, 1974, AFIPS Press, pp. 625- 635
- [52] Ramamoorthy, C.V. and Li, H.F., 'Pipeline Architecture', ACM Computing Surveys, Vol. 9, No. 1, March 1977, pp. 61-102
- [53] Ramamoorthy, C.V. and Wah, B.W., 'An Optimal Algorithm for Scheduling Requests on Interleaved Memories for a Pipelined Processor', IEEE Trans. on Computers, Vol. C-30, No. 10, October 1981, pp. 787-799
- [54] Riseman, E.M. and Foster, C.C., 'The Inhibition of Potential Parallelism by Conditional Jumps', IEEE Trans. on Computers, Vol. C-21, No. 12, Dec. 1972, pp. 1405-1411
- [55] Russell, R.M., 'The CRAY-1 Computer System', Communications of the ACM, Vol. 21, No. 1, January 1978, pp. 63-72
- [56] Siewiorek, D.P., 'Architecture of Fault Tolerant Computers', IEEE Computer, Vol. 17, No. 8, August 1984, pp. 9-18
- [57] Smith, J.E., 'A Study of Branch Prediction Strategies', Proc. Eighth Annual International Symposium on Computer Architecture, May 1981, pp. 135-148
- [58] Thornton, J.E., Design of a Computer System, the Control Data 6600, Glenview, Illinois: Scott, Foresman and Co., 1970
- [59] Thurber, K.J., Large Scale Computer Architecture, Rochelle Park, New Jersey: Hayden Book Company, 1976

- [60] Tomasulo, R.M., 'An Efficient Algorithm for Exploiting Multiple Arithmetic Units', IBM Journal of Research and Development, Vol. 11, No. 1, January 1967, pp. 25-33
- [61] Treleaven, P.C., Brownbridge, D.R., Hopkins, R.P., 'Data Driven and Demand Driven Computer Architecture', ACM Computing Surveys, Vol. 14, No. 1, March 1982, pp. 93-143,
- [62] Watson, W.J., 'The TI ASC - A Highly Modular and Flexible Supercomputer Architecture', Proc. AFIPS 1972 Fall Joint Computer Conference, AFIPS Press, 1972, pp. 221-228
- [63] Whitby-Stevens, C., 'The Transputer', Proc. Twelfth Annual International Symposium on Computer Architecture, June 1985, pp. 292-300
- [64] Widdoes Jr., L.C. and Correll, S., 'The S-1 Project: Developing High Performance Digital Computers', Energy and Technology Review, 1979
- [65] Yau, S.S. and Fung, H.S., 'Associative Processor Architecture - A Survey', ACM Computing Surveys, Vol. 9, No. 1, March 1977, pp. 3-27

END

22.06.88

FIN