STRATEGIES FOR THE MODELLING AND SIMULATION OF ASYNCHRONOUS COMPUTER ARCHITECTURES

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering

September 1995

By
Georgios Theodoropoulos
Department of Computer Science

Contents

List of Tables

List of Figures

Abstract

Synchronous VLSI design is approaching a critical point, with clock distribution becoming an increasingly costly and complicated issue and power consumption rapidly emerging as a major concern. Asynchronous digital design styles promise to liberate VLSI systems from clock skew problems, offer the potential for low power and high performance and encourage a modular design philosophy which makes incremental technological migration a much easier task. The desire to exploit the potential advantages offered by asynchronous logic has recently fueled a revival of interest in asynchronous systems.

Modelling and simulation, being at the heart of digital system design, may perform a catalytic role in the quest for the realization of the potentials offered by asynchronous logic. Hence, the recurrence of interest in asynchronous design has been accompanied by an intense research activity aiming at developing techniques appropriate for modelling and simulating asynchronous systems.

Contributing to this effort, and motivated by the increasing debate regarding the potential use of CSP for this purpose, this thesis investigates the suitability of occam, a CSP-based programming language, for the modelling and simulation of complex asynchronous designs.

A modelling approach is introduced which aims to exploit the strong relationship between the semantics of occam and the structure and operation of asynchronous systems, as well as the parallelism inherent in asynchronous hardware to achieve the rapid development of asynchronous architectural simulation models,

which may be executed on transputer networks to achieve high performance.

The applicability and robustness of the approach is demonstrated by employing it to construct occarm, an occam model of the AMULET1 asynchronous microprocessor.

The distributed nature of the proposed modelling approach introduces the problem of maintaining temporal precision and ensure that the causality principle is not violated.

The thesis provides a quantitative analysis regarding the timing error introduced in the model if violations of the causality principle are permitted. It then introduces the Program Driven Synchronization Protocol, a novel conservative, deadlock avoidance synchronization technique for dealing with causality problems within the framework of the proposed modelling approach.

Monitoring, debugging, termination and load balancing issues are also discussed.

DECLARATION

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

All trademarks cited within this work are acknowledged by the author.

COPYRIGHT AND OWNERSHIP OF INTELLECTUAL PROPERTY RIGHTS

Preface

The author obtained a Diploma degree in Computer Engineering and Informatics from the University of Patras, Greece, in July 1989. He was subsequently awarded a scholarship for graduate studies in Computer Science by the University of Manchester, U.K. In October 1990, he joined the Department of Computer Science at the University of Manchester as a research student (ParSiFal group) and a year later he completed his MSc (Method II-by research). In October 1991 he joined the AMULET group at the Department of Computer Science, University of Manchester to pursue a Ph.D.

Στους γονεὶς μου, Κωνσταντὶνο και Χϱυσοὺλα,
για ὸλα ὸσα ϑυσὶασαν στη ζωὴ τους για μὲνα

Και στην Εὺη,
για την αγὰπη και την υποστὴϱιξη
που μου ὲδωσε ὸλα αυτὰ τα χϱὸνια

To my parents, Konstantinos and Chrysoula,
for everything they sacrificed in their life for me

And to Evi,
for the love and support she gave me all these years

"Η Ιθὰκη σ' ὲδωσε τ' ωραὶο ταξεὶδι.
Χωρὶς αυτὴν δὲν θὰβγαινες στο δρὸμο.
'Αλλα δὲν ὲχει να σε δὼσει πιὰ.
Κι' αν φτωχικὴ την βρεὶς, η Ιθὰκη δὲν σε γὲλασε.
'Ετσι σοφὸς που ὲγινες, με τὸση πεὶρα,
ὴδη θα το κατὰλαβες η Ιθὰκες τι σημαὶνουν."

Κ. Καβὰφης

"Ithaca gave you the lovely journey.
Without her you would never have set out.
But she has nothing more to give you.
And if you find her poor, Ithaca has not deceived you.
Wise as you will have become, so full of experience,
you must have understood already the meaning of Ithacas."

C. Cavafy.

Acknowledgements

I am grateful to my supervisor, Dr. J. V. Woods, for everything he did for me during the four years in which the research for this thesis was carried out.

Professor S. B. Furber, as the director of the AMULET group, has always been a source of inspiration; for this, for his advice and guidance during the initial stages of this research, and for patiently providing me with the facilities to carry out the research for this thesis I express my thanks.

I also wish to thank Dr. P. Capon, for allowing me access to the T-Rack, and for his invaluable assistance during the implementation of the occam systems required for the research presented in this thesis. Thanks are also due to all those who supported the T-Rack the last four years, allowing me to complete my research.

I would also like to thank all the members of the AMULET group for their support all these years. In particular, I wish to express my thanks to Rob Kelly, (now with ICL, Manchester), Paul Day, Nigel Paver and Steve Temple for always being prepared to discuss and provide answers to my questions regarding the peculiarities of the AMULET1 architecture; Rob's help during some crucial moments of the work was invaluable. David Jackson, Craig Farnsworth, Rahul Mehra, Rhod Davies, Jim Garside, Oleg Petlin, Steve Nicklin, Dave Gilbert, they all helped me in a number of occasions and I thank them all. Nigel also provided his Lapwing Court 'mansion', which proved an excellent environment for writing up this thesis, not least because of his strict "Ph.D policing"; my special thanks.

The research presented in this thesis was funded by the University of Manchester and by the Mpakalas Foundation, Athens, Greece; I gratefully acknowledge this support.

The many friends I made during my five year stay in Manchester have been an invaluable source of emotional support and have made the Ph.D a less stressful task to undertake and Manchester a great place to live; I thank them all. Eudokia and Vaso, with their parties and joyful lifestyles helped to release some of the frustration of the Ph.D work and made the time I spent with them at 42 Egerton Road, Fallowfield, unforgettable; for that and for taking good care of me, I wholeheartedly thank them. George Tsakogiannis has been a continuous source of support and encouragement ever since we met in Manchester, back in 1990; his support during some very crucial moments of the work for this thesis was catalytic for its continuation and completion. Wael Fahmi's presence in Manchester had always been a source of optimism and positiveness, a window of light in the dark surroundings; my feelings for him could not be possibly expressed in words.

I also wish to express my special thanks to Fenia, Aris, Eirini, Xrysa and Sofia, for their genuine interest in my wellbeing while abroad, and for making me feel at home every time I visited Greece.

My parents, Kostantinos and Chrysoula, and my sister Eleni, have been a constant source of moral and financial support, throughout my life. I could not possibly give back to my parents what they have lost in their effort to support me - I try to honour their sacrifice. I have dedicated this thesis to them with love, respect and gratitude.

Last but not least, my deepest love and gratitude to Evi, who has stood by me like nobody else all these difficult years, and has offered me her support, encouragement, love and life - despite the separation. I dedicate this thesis to her too, with all my heart.

Chapter 1

Introduction

1.1 Background

This thesis is concerned with methodologies and techniques to support the modelling and distributed simulation of asynchronous computer architectures. The research presented in this thesis took place in the period 1991-1994 and relates to work in asynchronous system design undertaken by the AMULET group at the Department of Computer Science, University of Manchester.

1.2 Motivation and Objectives

Clocked VLSI systems are approaching a critical point, due to certain deficiencies inherent in the synchronous operation. Asynchronous logic promises to provide the means to overcome these deficiencies and limitations of the synchronous VLSI design approach. Hence, recently, there has been a resurgence of interest in asynchronous design techniques.

The quest for the exploitation of the potential advantages offered by asynchronous logic has revealed a need for modelling and simulation techniques, which would be appropriate for the asynchronous design style. Thus, the recurrence of interest in asynchronous design has fueled intense research activity aiming to develop techniques appropriate for modelling and simulating asynchronous systems. CSP, in particular, has attracted the attention of many researchers as a potential notation for describing asynchronous behaviour. Contributing to the quest for modelling and simulation techniques suitable for asynchronous design, and motivated by the increasing debate regarding the potential of CSP for this purpose, the work described in this thesis investigates the suitability of occam, a CSP-based programming language, for the modelling and simulation of complex asynchronous systems.

1.3 Structure of the Thesis

The thesis comprises 10 chapters. Chapters 2, 3, 4 and the first part of chapter 5 provide a theoretical background to the areas directly related to the subject of the thesis, namely parallelism, simulation modelling, asynchronous systems and modelling techniques for asynchronous hardware respectively. The remainder of the thesis, namely the second part of chapter 5 and chapters 6, 7, 8 and 9, describe the author's work and contribution.

Chapter 2 provides a short introduction to parallelism as a natural path to the quest for high performance, with emphasis on the MIMD approach to parallel computation. The various parallel programming models are mentioned and the CSP model of computation is discussed. Finally, a more detailed description of the occam programming language and its associated processor, the transputer, is provided.

Chapter 3 deals with the issues of modelling and simulation. After a short introduction to modelling, the chapter concentrates on discrete event simulation modelling, and in particular, its distributed implementation. The various approaches for exploiting parallelism in simulation are mentioned, and the Logical Process Paradigm is described. The causality-related issues arising from the distributed nature of the Logical Process Paradigm are described and the techniques that have been developed to address these issues are discussed. The chapter concludes with a discussion on the role of modelling and simulation in digital system design.

Chapter 4 discusses issues related to asynchronous design techniques. After a short discussion on the nature and advantages of asynchronous logic, the chapter presents some of the most influential asynchronous design techniques and then proceeds to provide a more detailed description of the Micropipeline design approach. The final part of the chapter is occupied by a short description of the Micropipelined AMULET1 microprocessor.

The thesis then moves to the modelling of asynchronous systems. Chapter 5 provides an overview of existing notations and techniques; emphasis is placed on the techniques that employ CSP-like notations. Advocating the potential use of occam for the modelling and simulation of asynchronous systems, a methodology is introduced which employs occam to build parallel models of asynchronous systems based on the Micropipeline approach. This neglects causality problems, thus allowing timing errors to occur. It is argued however that the timing inaccuracy will be acceptable.

Chapter 6 describes how the modelling approach introduced in chapter 5, has been employed to build an occam model of the AMULET1 asynchronous microprocessor. Emphasis is given to the aspects of the modelling that proved to be complex. The chapter also includes a more detailed description of the operation of the AMULET1 processor, as this defines the functionality of the model as well.

Chapter 7 addresses issues related to the execution of the occam model on a computer system. Two environments are presented, for the execution of the model on a single and multiple transputers respectively. Monitoring, termination, mapping and load balancing issues are discussed.

Chapter 8 presents a validation of the occam model, providing quantitative results and analysis regarding both, the accuracy and performance of the model.

Chapter 9 introduces the "Program Driven Synchronization Protocol" (PDSP), a novel approach for eliminating the timing problems raised by the distributed nature of the proposed modelling philosophy. The concept of "Instruction Lookahead" is introduced and a set of rules are presented which specify the behaviour of processes in the occam model so that timing accuracy is ensured. The application of PDSP onto the occam model of AMULET1 is then described. Finally, some performance results are given.

Chapter 10 epitomizes the conclusions drawn by the research presented in the thesis, and indicates a number of areas where further research and development work is possible.

1.3.1 Related Publications

Different aspects of the research work presented in this thesis have been presented in the AMULET Modelling Workshop (Windermere, Cumbria, England) [Theo94], the World Transputer Congress 1994 (Como, Italy) [Theo94a], the European Simulation Symposium 1994 (Istanbul, Turkey) [Theo94b], the Eurosim Congress 1995 (Vienna, Austria) [Theo95] the World Transputer Congress 1995 (Harrogate, England) [Theo95a] and the 4th Euromicro Workshop On Parallel And Distributed Processing (Braga, Portugal) [Theo95b].

Chapter 2

The Quest for High Performance

2.1 Introduction

The principles of computer organization have traditionally been based upon the processing model described by John von Neumann in the late 1940s, whereby a computer comprises a single processing unit connected to a single[1] memory in which both, the program code and the data to be operated upon are stored. This is commonly referred to as the von Neumann model of computation. The operation of the system consists of sequential fetches of instructions and their operands from memory, which are then executed, with the result of the execution written back to memory. The memory addresses of all instructions and data involved in a computation as well as the instructions their operands and the produced results are communicated over the single path that connects the processor and the memory; Backus called this path as the "von Neumann bottleneck" [Back78].

Since the first commercial, general purpose von Neumann computers were commissioned in the early 1950s, there has been an ever increasing demand for

[1]The basic memory model, which uses the same physical memory to save both instructions and data is referred to as the "Princeton Architecture".

more computational power and speed. Traditionally, this demand has been satisfied by enchancing the performance of sequential computers through technological advances and architectural innovations.

The progression from electromechanical relays, vacuum tubes, William's tubes and magnetic drums, to transistor switching devices and solid state memories, to integrated circuits and eventually to VLSI semiconductor devices has had a dramatic impact on computational speed. Reduced physical size and shorter device switching times have resulted in digital systems that can be driven at extremely high clock rates (e.g. the 21164 EV-5, successor of the Dec's Alpha processor, is expected to run at a speed of 300MHz [Gwen94]). Optical technology promises to reduce gate delays to picoseconds while increasing the bandwidth and the numbers of interconnections [Huan90].

Advances in technology have been accompanied by architectural novelties. These include the use of cached store hierarchies, virtual address spaces, arithmetic optimizations and more efficient instruction sets. Techniques to overcome the von Neumann bottleneck have also been devised; these include the "Harvard Architecture" which uses separate instruction and data memories and memory interleaving, which allows concurrent access to several independent memory modules [Kraf79] [Ibbe82] [Furb89] [Heud92].

2.2 Bit and Instruction Level Parallelism

A technique which has made a significant contribution to the improvement of the performance of von Neumann architectures has been the exploitation of parallelism at bit and instruction levels.

Bit-parallel arithmetic was the first technique to be employed as hardware components became less expensive.

Bit parallelism was followed by the introduction of multiple functional units (*stages*) which could operate in an overlapped manner. These stages are connected one to the next to form a pipeline; instructions enter at one end of the pipeline and travel through consecutive stages to exit at the other end. This arrangement allows an instruction to be (pre)fetched and decoded while the previous instruction is being executed. The execution unit may also be divided into a number of stages, with each stage performing a different operation contributing to the execution of an instruction. Several advanced architectural techniques have been devised to optimize pipelined operation; these include branching schemes (e.g. delayed branch), advanced scheduling techniques, register bank management techniques, superpipelining, superscalar and very long instruction word (VLIW) systems [Henn91]. Superpipelined machines employ "deeper" pipelines, while superscalar machines issue more than one instruction per clock cycle in the pipeline; in VLIW systems the compiler finds operations which can be issued in the pipeline together and creates a single instruction containing those operations. In general, the utilization of multiple pipelined units can reduce the average execution time per instruction and improve throughput significantly; however, data and control dependencies may cause considerable delays [Kogg81]. Traditionally, pipelines have been driven by a common external clock. Recently unconventional techniques have been developed which allow the asynchronous operation of pipeline stages; these techniques are at the core of the research work described in this thesis and are discussed in detail in chapter 4.

2.3 Reduced Instruction Set Computers

A milestone in computer architecture research was the development of Reduced Instruction Set Computers (RISCs).

The quest for high performance and the aim to reduce the "semantic gap" between hardware and high level programs had generated a trend towards increasingly sophisticated and complex hardware and instruction sets [Furb89]. Against this background of increasing complexity the concept of RISC was introduced in 1980, advocating simplicity and efficiency [Patt80]. RISC systems implement small instruction sets using simple and fixed instruction formats on load/store architectures to achieve fast and single cycle execution [Stal88]. They employ pipelining and hardwired (instead of microcoded) control and rely on optimising compilers and large cache memories to maximise the use of registers and minimize references to main memory. The underlying principles of the RISC approach had already been laid down by IBM in their 801 computer, developed in late 1970s at IBM's Thomas J Watson Research Center [Radi83]; the first prototype VLSI RISC processors were developed in early 1980s at the University of California, Berkeley [Kate85] and at Stanford University [Henn81]. Since then, and amidst an ever lasting controversy on the issue of system complexity [Whar92] [Alli92], several commercial powerful RISC processors have been developed, including the VL86C010 Acorn RISC Machine (ARM) [Furb89] [VLSI90], the MB86900 SUN SPARC [Fuji87] [Sun87], the MIPS R2000 [Kane87], AT&T's CRISP [Ditz87], AMD's Am29000 [AMD87], Hewlett-Packard's HPPA [HP86] and Motorola's M88000 [Dobb88]; comprehensive surveys of VLSI implementations of RISC architectures may be found in [Furb89] and [Heud92].

2.4 The Limits of Sequential Computation

Technological and architectural advances have yielded a dramatic rise in the performance of von Neumann machines and computational speed increases of an order of magnitude every five years have been witnessed. However, this rate of improvement can not be sustained due to fundamental limitations imposed on VLSI technology by the laws of physics [Lind93] [Prep94]. One such limitation is the speed at which electrical signals can be transmitted over the physical links which connect the circuit components; this speed cannot be greater than the speed of light (i.e. about 0.3 metres per nanosecond) [Russ78]. Miniaturization techniques aim at high package densities and a high degree of integration to reduce distances between components, but are limited by the heat dissipation and the quantum effects at sub-micron levels [Mead80]. These two fundamental constraints impose an ultimate limit to the maximum theoretical performance that can be achieved by a single von Neumann computer; this performance has been estimated to be around 3 Gigaflops (3 billion floating point operations per second) [Wile87].

2.5 Parallel Computer Architectures

Despite the dramatic rise in their performance, sequential von Neumann computers still can not offer the processing rates required for the solution of a wide range of applications (the so called "Grand Challenge" problems) [Fox89] [Lazo93] [Kung94]. This has led to the introduction of architectural concepts which attempt to take advantage of the high level, algorithmic or data-set, parallelism inherent in many problems. These concepts call for the utilization of multiple processing elements which can operate in parallel [Hwan84] [Hock88] [Dunc90] [Mold93] .

Traditionally, parallel computer architectures are classified as either SIMD (Single Instruction Multiple Data Streams) or MIMD (Multiple Instruction Multiple Data Streams), following Flynn's taxonomy [Flyn72]. This taxonomy defines two more classes of architectures, namely SISD (Single Instruction Single Data Streams) and MISD (Multiple Instruction Single Data Streams). SISD refers to conventional, serial von Neumann machines; in these computers at any instant, one stream of instructions (and therefore, only one instruction processing unit) operates on a single data stream. MISD would involve multiple processing units applying different instructions to a single datum; this theoretical possibility is generally deemed impractical.

2.5.1 SIMD

Two major classes of SIMD machines exist, namely vector processors and array processors.

Vector processors utilize multiple pipelined processing elements to apply identical arithmetic operations to a data stream linearly organised in vector registers. Vectorising compilers are employed to replace blocks of sequential code by vector instructions. Examples of successful commercial vector machines are the CRAY-1 [Russ78], the Control Data Corporation CYBER 200 series [Linc82] and the Fujitsu VP-200 [Miur84].

Array processors typically employ a central control unit and multiple processing elements which operate in parallel applying the same instruction sequence on arrays of data in a *synchronized* fashion. Array machines can achieve high performance rates with suitable problem classes where the same operation must be performed on different data sample points. Generally, parallelism must be expressed explicitly by the programmer, although compiling techniques can also be used for this purpose. Examples of array processors are the ILLIAC IV [Hord82], the Burroughs Scientific Processor (BSP) [Aust79], the ICL Distributed Array Processor (DAP) [Redd79], the Goodyear Aerospace MPP [Batc80] and the Connection Machine [Hill85].

SIMD machines are well suited to data parallel applications such as weather forecasting, image processing, finite state analysis and general linear algebra problems.

2.5.2 MIMD

MIMD architectures employ multiple connected processors which can execute independent instruction streams. MIMD computers are *asynchronous* systems characterized by decentralized hardware control. They offer greater flexibility than their SIMD counterparts and they support higher level parallelism (subprogram and task levels) which can be exploited by *divide and conquer* algorithms; a large problem is split into a number of sub-tasks, which can then be executed concurrently on different processors.

The degree of coupling between processors categorizes MIMD machines as either tightly or loosely coupled. In tightly coupled (or shared memory) systems, processors communicate and synchronize through a global shared main memory. In loosely coupled (or distributed memory) systems, each processor has its own local store and communicates with the other processors by exchanging messages via a network[2]. *Virtual shared memory* systems have also been developed, wherein the memory is physically distributed but it is logically viewed by the programmers as a single global shared memory: these systems include DDM [Warr88] and KSR-1 [KSR].

2.5.2.1 Shared Memory MIMD Architectures

In shared memory architectures, performance may be degraded due to store contention problems which may occur if more than one processor requires access to the store simultaneously. To reduce this possibility, multiple banks of store may be used. Another technique uses local cache memories to reduce accesses to the global memory. This introduces the issue of maintaining coherency among the multiple copies of the same data that may exist in various processors' caches at a given time. Cache coherency protocols, implemented in hardware or software,

[2] Distributed memory architectures are also referred to as message-passing systems.

may solve this problem but usually reduce the performance of the system [Hill89] [Sten90].

Various alternatives for connecting multiple processors to shared memory have been proposed. Small machines with relatively few processors typically feature a time-shared data bus; processors contest control of this to access the store. Bus saturation problems make this scheme ineffective for large processor numbers. Flexible Corporation's Flex/32, the Encore Computer's Multimax and the Sequent Balance machines [Hock88] are examples of bus based architectures. Crossbar interconnection technology uses a crossbar switch of n^2 crosspoints to connect n processors to n memories. Carnegie Mellon multi-mini-processor (C.mmp) [Wulf72] was based on this interconnection scheme. Power, pinout, and size considerations make fully connected networks very expensive because the complexity grows as the square of the number of processors and memories. Multistage interconnection networks such as Delta and Omega networks [Pate79] provide the same connectivity as crossbar networks but at reduced cost[3]; an example of a multilevel switching network based machine is the Illinois Cedar computer [Gajs83] which uses an Omega network to connect clusters of processors to global memory modules.

Another problem associated with shared memory architectures is the issue of *mutual exclusion*, namely preventing a task from accessing a shared data structure while this is being modified by a different parallel task. Signals [Wirt77], semaphores [Dijk68], conditional critical regions [Hoar72] [Hans73], and monitors [Hoar74] are schemes that have been developed to solve the mutual exclusion problem. Languages based on these concepts have also been developed including Ada [Barn89] and Modula [Wirt77].

[3] Delta networks for instance have a cost logarithmic to the number of inputs.

2.5.2.2 Distributed Memory MIMD Architectures

Message passing MIMD machines have principally been constructed in an effort to provide architectures with the potential to scale up into systems consisting of many thousands of processors.

In distributed memory architectures[4], processors are connected by an interconnection network; bus-based configurations also exist (e.g. the PRINGLE computer [Hock88]) but are not common.

The network that provides for the connection of the processors in a message passing system can be either static or reconfigurable. In the first case, the connections between the processors are fixed and permanent. In static networks, communication between non-neighbouring processors can result in significant latency as data is queued and forwarded by intermediate nodes to reach its destination. Various interconnection network topologies have been proposed to support expandability and scalability and to minimize latency for certain classes of problems [Sieg85] [Reed87]; these include rings, grid based networks such as meshes, cylinders, toroids, trees, cube connected cycles, shuffle exchange networks, and hypercubes.

Reconfigurable topology architectures provide programmable switches that allow users to select a logical topology which matches a particular application's communication pattern. Usually the desired configuration is arranged in advance, though dynamic reconfiguration is also an available option [Murt91].

Examples of commercial distributed memory MIMD machines are the BBN Butterfly [Hock88], the Cosmic Cube [Seit85], the ChiP computer [Snyd82] and the Intel iPSC [Inte86].

Recently, some less conventional MIMD based architectural paradigms have

[4]The term typically applies to systems in which the nodes are closely connected via very high speed communication links. Physically distributed computers connected by slow message passing networks (e.g. ARPANET) are usually excluded.

been developed. Dataflow architectures [Gurd85] and graph reduction machines [Trel82] are examples of such paradigms.

2.5.3 Parallel Programming Models and Languages

The von Neumann model of computation is typically characterized as *control driven*. There is a single thread of control normally passed sequentially from instruction to instruction thus determining the sequence in which instructions are executed. The control driven model of computation is supported by conventional imperative languages such as C [Kern88].

The advent of parallel architectures has resulted in a strong interest in different computation models which have the potential for parallel execution and new programming styles and languages to support them.

In the *data driven* model of computation, an instruction may be executed as soon as the necessary data are available, without taking into account its textual position in the program. This model is supported by *single assignment* languages such as SISAL [Böhm91] and LUCID [Asch77]. Data flow machines constitute implementations of the data driven model of computation.

In the *demand driven* model, an operation is performed only when its result is required. *Functional languages* [Glas84] [Read89] such as HASKELL [Huda90] are well suited for this model, while graph reduction architectures provide for their efficient implementation.

Logic languages are based on *pattern driven* models of computation [Alma89]. In Prolog [Cloc81] for instance, the execution of a program consists of a search for facts (patterns) which satisfy a given query; the facts are organized to form a search tree which may be searched in parallel (using AND-parallelism or OR-parallelism).

The three aforementioned models of computation support fine grain parallelism which is the responsibility of the compiler to exploit. However, the predominant computational model for MIMD architectures, is based upon the concept of concurrent processes that communicate and synchronize through the exchange of messages (message passing); this model is well suited to distributed memory machines though it can also be implemented on shared memory systems. The process based model of computation exploits the *algorithmic* parallelism of programs and calls for their partitioning into a number of different processes which perform different parts of the overall algorithm[5]. The parallel processes may then be mapped onto different processors of the parallel architecture. The detection of algorithmic parallelism and the mapping of the resulting partitioning onto the multiple processors of an MIMD machine are two major problems associated with the process based model of computation. Ideally, the latter task should be performed automatically, though in general it is the programmer's responsibility; chapter 7 discusses how this problem has been tackled within the context of the research presented in this thesis.

Various process based programming models have been proposed, including PLITS (Programming Language in the Sky) [Feld79], Linda [Ahuj86], and CSP (Communicating Sequential Processes) [Hoar78] [Hoar85]. These differ in the naming convention they use to refer to sender and receiver processes and the semantics of their communication.

2.5.3.1 Communicating Sequential Processes

The theoretical model of Communicating Sequential Processes is one of the most elegant schemes which have been proposed for process based parallel computation.

Within the framework of CSP, a program is a collection of sequential processes

[5]Usually the process structure of an algorithm implemented using message passing is represented by a graph whose nodes represent the processes and edges represent the communication paths between them.

which execute asynchronously and concurrently and communicate by exchanging messages through *channels*.

The communication is unbuffered, point-to-point and synchronous. A communication requires a *rendezvous* between the sender and the receiver processes and will take place only if both these processes have requested participation in such an activity; otherwise the process that arrives at the rendezvous first has to block until the corresponding action is reached at the other end of the channel.

A guarded input construct [Dijk75] is available which provides the means for a receiver process arbitrarily to select to communicate via one of several input channels depending on the readiness or otherwise of these channels; the choice is nondeterministic.

2.6 Occam and the Transputer

With the aim of providing a practical realization of the CSP model of computation, in early 1980s Inmos Limited developed the *occam* programming language [Inmo88] and the *transputer* [Inmo86] [Inmo88a], a microprocessor designed to support the execution of occam.

2.6.1 The Occam Programming Language

The basic unit of the occam language[6] is the *process*. The concept of the process can be viewed at many levels within an occam program, the lowest being the command level. Occam programs are built from three *primitive* processes namely

[6]Within the context of this thesis, the term occam refers to the occam2 language. This is a development of an earlier occam language known as occam1 or proto-occam. The main difference between the two occam variants is that occam2 allows the transmission of structured data types along channels, by providing extended channel protocols. Recently, Inmos developed another variant of occam, referred to as occam3, which supports records and user-defiable data types [Inmo92]. Tutorial introductions of occam2 may be found in [Kerr87] [Poun87] [Burn88] [Gold88] [Dows88] and [Gall90].

assignment, *input* and *output*:

- $v := e$: assign expression e to variable v
- $c\ !\ e$: output expression e to channel c (send)
- $c\ ?\ v$: input variable v from channel c (receive)

Channels are used to enable the exchange of messages between concurrently executing processes. The semantics of occam channel communications are based on CSP. A communication through a channel is unidirectional, unbuffered, point-to-point, and synchronous; as in CSP, a sender and a receiver have to establish a rendezvous in order to communicate.

The primitive processes may be combined to form *constructs*:

- *IF* : conditional
- *WHILE* : iterative
- *SEQ* : sequential
- *PAR* : parallel
- *ALT* : alternative

A construct is itself a process and may be used as a component of another construct. The scope of these constructs is indicated in the text of the program by indentation (a single unit of indentation being two spaces). The execution of a compound construct terminates when all the processes within it (i.e. constructs and primitive processes) have terminated.

IF and WHILE constructs are the standard conditional and iterative commands respectively, encountered in all imperative languages.

```
SEQ
  P1
  P2
  PAR
    P3
    SEQ
      P4
      P5
    P6
  P7
:
```

Figure 2.1: The Use of the Occam SEQ and PAR Constructs

2.6.1.1 The SEQ and PAR Constructs

The mode of process execution is declared using the SEQ and PAR constructs. Processes which are grouped using SEQ are executed in sequential, textual order. Processes grouped by means of PAR, are not restricted to any specific order of execution and may thus be executed in parallel. SEQ and PAR constructs may be combined to allow the specification of arbitrarily complex execution orderings; the only restriction is adherence to the communication rules (e.g. parallel output to the same channel is illegal).

In figure 2.1, processes P1 and P2 will execute sequentially. Upon completion of P2 the PAR construct is activated which allows processes P3, P6 and any one of the processes P4 and P5 to execute concurrently; P4 and P5 have to execute sequentially. Upon the termination of the PAR construct, P7 is executed.

Occam allows the *replication* of the SEQ and PAR constructs by providing an extension of their syntax. A replicated SEQ corresponds to the FOR command, encountered in most imperative languages. The parallel replicator is more interesting, for with it arrays of parallel, identical processes may be specified.

Priorities among parallel processes executing on the same processor may be enforced by the use of the PRI operator (i.e. PRI PAR); the precedence in

```
BOOL ready:
CHAN OF BYTE ch1, ch2:
BYTE any:

ALT
  ch1 ? any                --input guard of process P1
    P1
  ready=TRUE & ch2 ? any   --input guard of process P2
    P2
:
```

Figure 2.2: The Occam ALT Construct

this case is declared by the textual order of the processes within the PRI PAR construct.

The transputer implementation of occam supports two levels of priority, namely *low* and *high*. Low priority processes are time-sliced and are executed only when there is no active high priority processes. High priority processes are not time sliced. Both, low and high priority processes are descheduled each time they need to wait on a channel or a timer communication (see section 2.6.1.3). By default, occam processes execute at low priority.

2.6.1.2 The ALT Construct

The ALT construct implements the guarded input command of the CSP model. A guard can be either an input process or an input clause accompanied by a boolean condition (figure 2.2).

ALT enables a receiver to select for execution one of several alternative channel input processes; this allows the receiver to communicate with more than one sender process, receiving randomly ordered messages. The input process selected for execution is the one whose boolean condition, if any, is TRUE and the corresponding sender process at the other end of the channel is first

```
VAL Delay IS 15625:  --Constant definition
TIMER Clock:         --Timer channel declaration
INT Now:

SEQ
  Clock ? Now --Read current clock value
  Clock ? AFTER (Now PLUS Delay) --Deschedule
  --Resume Operation
:
```

Figure 2.3: Introducing Delays with Occam Timers

ready to communicate; if more than one ready sender exists, an arbitrary, non-deterministic choice is made.

A prioritized ALT is also provided (i.e. PRI ALT), in which case the readiness of the guards is examined in their textual order.

2.6.1.3 Timers

One of the original target areas of occam was in embedded systems applications, where support for real time control is required. This support is provided in occam by means of *timers*. Timers are syntactically treated as communication channels which can provide only input. The value returned is the current time which is represented as an integer value. The duration of the clock tick depends on the priority level of the process wherein the timer is invoked. In transputer implementations of occam, for low priority processes, each clock tick represents 64 microseconds; for high priority processes, the clock is incremented every 1 microsecond.

Timers can be used to control the temporal behaviour of occam programs by forcing processes to delay their execution for a pre-determined period. This is illustrated in figure 2.3. The timer input statement, when used in conjunction with the AFTER keyword, causes the timer input to be held up until the current clock reaches the value specified in the AFTER clause. During this period, the

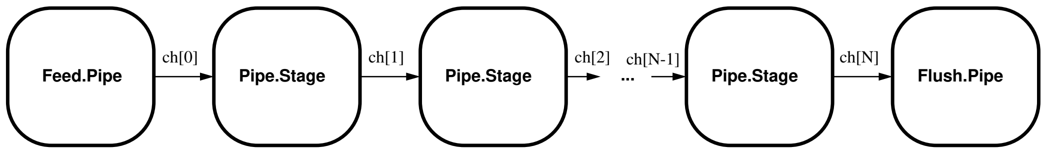
```
VAL Time.Out IS 15625:  --Constant definition
TIMER Clock:            --Timer channel declaration
INT Now:
BYTE any:
SEQ
  Clock ? Now   --Read current clock value
  WHILE (NOT end)
    SEQ
      ALT
        ch1? any  --Poll channel ch1
          P1

        Clock ? AFTER (Now PLUS Time.Out) -- Timeout?
          P2
:
```

Figure 2.4: Programming Timeout Behaviour

process remains descheduled. In general, scheduling delays introduced by the transputer's scheduler (see section 2.6.2) will slightly extend the period that the process remains inactive [Gall90]. Furthermore, an extra delay is imposed by the latency of the process queue of the processor, namely the time from when the process is rescheduled to the time at which reaches the front of the queue and starts processing; the maximum latency is estimated to be $(2n - 2) * time\_slice$, where n is the number of processes in the queue when the timer process is rescheduled [Mitc90]. Therefore, the period that a process may be caused to delay its execution can only be approximate and non-deterministic and is generally greater than the delay specified in the AFTER clause.

Used within an ALT construct, a timer may generate time-out behaviour and thus prevent deadlock situations that might be caused by processes waiting for input on unresponsive channels. Figure 2.4 depicts a process with time-out behaviour; if channel $ch1$ fires before the $Time.Out$ period has elapsed, $P1$ will be executed, otherwise the timer channel will fire and process $P2$ will be executed.




```
PROC Pipeline(...)
  VAL INT N IS ...:       --Size of pipeline
  [N+1] CHAN OF INT ch:  --Array of channels
  INT i:
  ...
  ---------------------------------------
  --Definition of the Feed.Pipe process
  ---------------------------------------
  PROC Feed.Pipe(CHAN OF INT To.Pipe)
    INT x:
    SEQ
      x:=0
      WHILE TRUE
        SEQ
          To.Pipe ! x
          x:=x+1
  :
  ---------------------------------------
  --Definition of the Flush.Pipe process
  ---------------------------------------
  PROC Flush.Pipe(CHAN OF INT From.Pipe)
    INT x:
    SEQ
      WHILE TRUE
        SEQ
          From.Pipe ? x
  :
  ---------------------------------------
  --Definition of the Pipe.Stage process
  ---------------------------------------
  PROC Pipe.Stage(CHAN OF INT From.Previous, To.Next)
    INT x:
    SEQ
      WHILE TRUE
        SEQ
          From.Previous ? x
          ...
          To.Next ! x
  :
  ---------------------------------------
  --Body of the calling Pipeline process
  ---------------------------------------
  SEQ
    PAR
      Feed.Pipe(ch[0])
      PAR i=0 FOR N
        Pipe.Stage(ch[i], ch[i+1])
      Flush.Pipe(ch[N])
:
```

Figure 2.5: An Example Occam Program

2.6.1.4 Functions and Procedures

As in all modular imperative languages, occam allows the definition of functions (FUNCTION) and procedures (PROC). Procedures encapsulate occam processes; a call to a procedure instantiates the occam process defined by the body of the procedure.

To illustrate the structure and philosophy of occam systems, a small example program is presented in figure 2.5.

Three different processes namely *Feed.Pipe*, *Flush.Pipe* and *Pipe.Stage* are defined as occam procedures. A replicated PAR construct is used to create a pipeline of N processes of type *Pipe.Stage*. Each process in the pipeline inputs data from the preceding process and outputs data to the succeeding process after performing some processing on the data. The pipeline may thus produce an overlapped operation with each component process of the replicated PAR executing concurrently with every other component process, input and output being automatically synchronized between processes, and a stream of data passing through the pipeline. A one-dimensional array of channels *ch* is declared, along the elements of which the processes communicate.

2.6.2 The Transputer

To provide the means for the efficient execution of occam based parallel applications, Inmos accompanied the development of occam with the design and implementation of a new family of microprocessors, collectively referred to as the *transputer*. The most important classes of the family have been the T400 series and the T800 series; recently Inmos have developed a more advanced type of transputer, namely the T9000.

The transputer is distinguished from conventional microprocessors in that it is particularly designed as a building block for distributed memory MIMD systems.

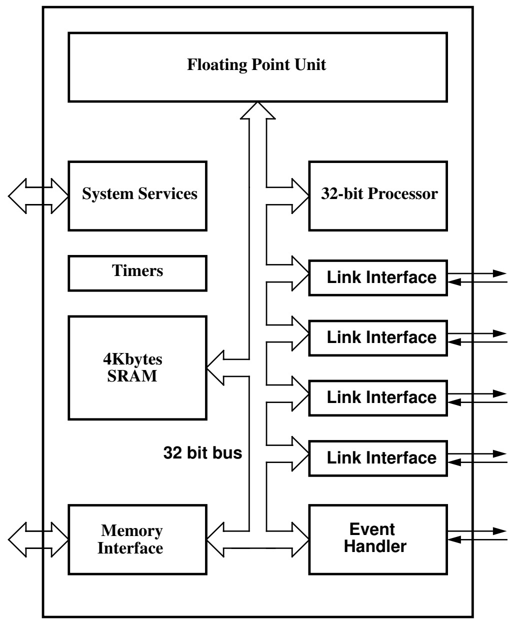


Figure 2.6: The Architecture of the T800 Transputer

Such a system can be constructed from a collection of transputers which operate concurrently and communicate through serial communication links.

In the transputer, INMOS have implemented on a single chip the main components of a traditional von Neumann computer, while at the same time providing a high level of support for a concurrent, process-based view of computation.

The processing part of the transputer is a 32-bit processor which executes a small[7] range of instructions and addressing modes at a rate of up to 10 MIPS (for 20MHz implementations). The T800 (figure 2.6) features a microcoded 64 bit floating point unit, providing a sustainable performance of 1-2 MFLOPS. The devices of the T400 series do not have a floating point unit and give an estimated performance of 0.1 MFLOPS.

The transputer has two (T400) to four (T800) Kbytes of high speed, on-board static RAM which is used as local memory. Off-chip memory may be accessed

[7]The transputer is often classified as a RISC, although there is a controversy on the correctness of this classification.

via a 32-bit wide external memory interface which allow data transfer rates of more than 25 Mbytes per second. In multi-transputer systems, each transputer has sole use of its own on-chip and off-chip memory and thus does not have to compete with the other transputers for common memory accesses.

The transputer incorporates four communication links. Each link can transfer data at over 1 Mbytes/s with automatic handshaking synchronization in each direction, providing a bidirectional, point-to-point, synchronous connection between transputers. A transputer may be linked to four other transputers. In this way networks of transputers of various sizes and topologies may be built up. Each transputer in a network operates as an independent unit communicating as and when necessary with the other transputers to which it is linked.

All transputer components operate concurrently; each of the four links and the floating-point coprocessor (on the T800) can all perform useful work while the processor is executing other instructions.

The transputer is closely integrated with the occam programming language: occam reflects the concurrency found in the transputer and provides for control of the transputer hardware, while the transputer provides efficient support for the occam model of concurrency and communication.

In occam, concurrency may be specified between transputers or indeed within a single transputer. To support internal concurrency, the transputer includes a process queue, with a microcoded scheduler which enables any number of concurrent processes to be executed together sharing the processor time. The microcoded scheduler, combined with a minimal context for each active process, achieve very rapid process switching (typically less than one microsecond).

Examples of T400/T800 transputer based machines are the ESPRIT Supernode Machine [Hock88], the Meiko Computing Surface [Bott86], the Parsytec's GCel and XPlorer machines [Pars], and the T-Rack [Capo86], a 64 transputer machine which has been implemented at the University of Manchester (the T-Rack is described in chapter 7).

2.6.2.1 Configuring Occam Programs

At the top level, an occam program is a network of concurrent communicating processes, defined as occam procedures. These procedures are independent, without reference to any shared program variables or data structures; any required communication takes place by exchanging messages over channels.

The top level occam processes may run time-sliced on a single transputer or be distributed over a network of transputers to achieve real parallelism.

In the first case, the communication channels are realized by internal (or *soft*) links. These are implemented by means of a status word in memory which may be accessed by both the sending and the receiving processes. Consequently, the number of internal occam channels is limited only by the size of local memory.

For multi-transputer configurations, channels are placed on one of the four physical links (*hard* channels) of the transputer. Hardware links can support only a single occam channel in each direction; if more than one occam process needs to access the same link, software multiplexing is required.

In occam, the allocation of processes to processors and channels to hardware links in the network is the responsibility of the programmer, who must devise and explicitly specify an appropriate mapping scheme. This allocation is static, though, in principle it is possible to use occam to code mechanisms for dynamic placement and running of processes [Waym89].

Occam includes a notation, the occam configuration language, which provides the means for specifying the adopted mapping of the occam program onto the transputer network[8] [Inmo91a]. A textual configuration description created using

[8]Providing two different languages, one for the code that describes the functionality of different processes and one to specify the configuration of the program, is a typical technique

this language basically describes the top level of the occam program, annotated with enough information to map the program on to a transputer network. This information includes:

- The type of processors in the system.
- Which processes will execute on which processor; one or more processes may execute on a single transputer.
- The mapping of occam channels to transputer links for inter-transputer communications

The considerations that need to be taken into account by the programmer for the mapping of occam programs onto transputer networks are discussed in chapter 7.

2.6.2.2 The T9000 Transputer

To alleviate the connectivity limitations and the channel mapping problems imposed by the four links of the conventional transputer, Inmos have developed a new family of transputer devices, namely the T9000 processor and the C104 transputer link switch [Inmo91].

The T9000 transputer still possesses four communication links, however it also incorporates a *Virtual Channel Processor* which allows the (hardware) multiplexing of an arbitrary number of occam channels onto each of the four links [Inmo93]. The C104 link switch is a wormhole routing device which can direct a packet arriving on any of its 32 input links to one of the 32 output links, based on the packet's header. The combination of T9000 transputers with possibly multistage

---

in distributed systems which require the the explicit specification of configuration by the programmer. This approach is closely related to the ideas of programming-in-the-large and the use of a module interconnection language [DeRe76].

C104 switches enables the construction of arbitrarily complex networks without the need for software harnesses to store and forward messages [Inmo93a].

T9000 also exhibits enchanced performance characteristics, operating at 30MHz and yielding an instruction throughput of about 50 MIPS and an estimated floating point performance of 5 MFLOPS.

Recently, some T9000 based machines have been commercially available, including the Parsys SN9400SP, SN9500 and SN9800 systems [Pars95].

2.7 Summary

This chapter has presented a short overview of parallelism, as an approach to satisfy the need for high performance. Emphasis was given on the process-based model of computation supported by the occam programming language and the associated processor, the transputer.

The next chapter deals with modelling and simulation; emphasis is placed on distributed simulation, which is a particular instance of parallel processing.

Chapter 3

Modelling and Simulation

3.1 Introduction

Since the dawn of civilization, people have tried to understand the principles and behaviour of systems in their environment, with which they have been in constant contact. An essential tool to this endeavour has been the development of *models*, namely simplified representations of the systems under consideration:

> "*For everything that exists there are three things through which knowledge about it must come; the knowledge itself is a fourth; and as a fifth we must posit the actual object of knowledge which is the true reality. We have then:- first, a name; second, a description; third an image; fourth knowledge of the object ... There is, for example, something called a circle, whose name is the very word I just now uttered. In the second place there is a description of it, made up of nouns and verbs. The description of the object whose name is round and circumference and circle would be: that which has everywhere the same distance between the extremities and the middle. In the third place, there is the object which is drawn and erased and turned on the lathe and destroyed*

- processes which the real circle, in relation to which these other circles exist, can in no wise suffer, being different from them. In the fourth place there are knowledge and understanding and correct opinion about them, all of which must be posited as one thing more, inasmuch as it is found not in sounds nor in the shapes of bodies but in soul, whereby it manifestly differs in nature both, from the real circle and from the aforesaid three. Of these, understanding approaches nearest to the fifth in kinship and likeness, while the others are more distant . . . Every circle drawn or turned on a lathe in practice abounds in the opposite to the fifth-for it everywhere touches the straight, while the real circle, we maintain, contains in itself neither more nor less of the opposite nature. The name, we maintain, is in no case stable; there is nothing to prevent the things now called round from being called straight, and the straight round; and those who transpose them and use them in the opposite way will find them no less stable than they are now." Plato, Epistle, vii, 342A-343B.

A model is intended to:

- Act as a communication vehicle, making available a description of the behaviour of a system.
- Enable users to gain insight and understanding regarding the behaviour of the system and develop theories that account for that behaviour.
- Provide the means for the analysis and evaluation of the system as well as the prediction of its future behaviour.

Figure 3.1 depicts a taxonomy of the different types of models [Neel87].

Mathematical modelling employs mathematical notations and mathematical equations to produce a description of the system.

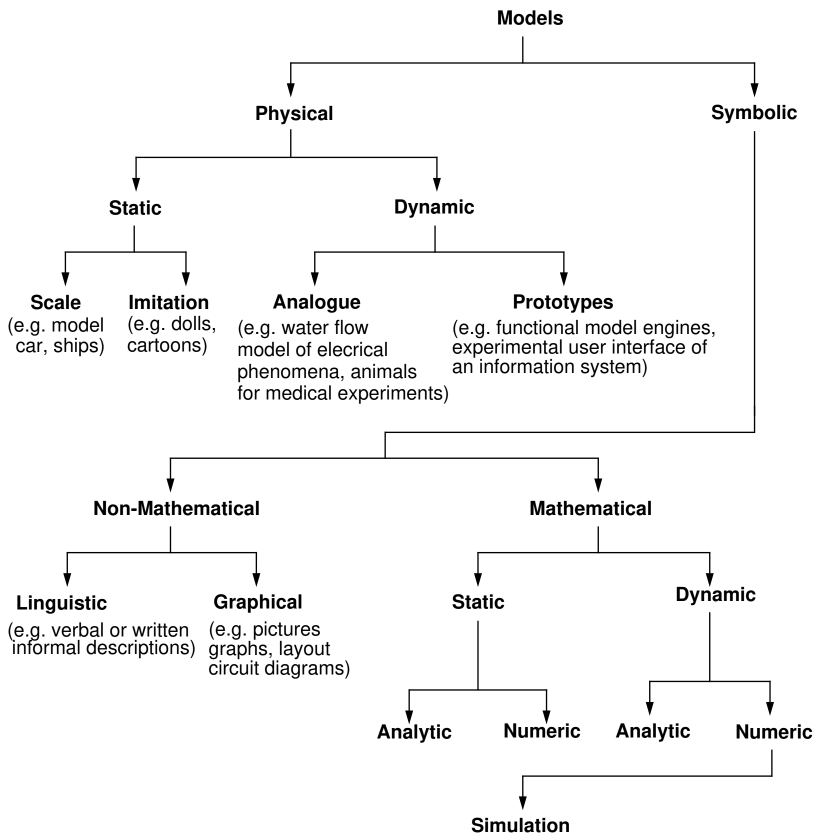


Figure 3.1: A Taxonomy of Models

A system is characterized as static, if its state[1] does not change but remains stable over time (i.e. the system is in equilibrium). The state of a static system is typically expressed mathematically using algebraic equations.

Systems with dynamic behaviour, i.e. systems whose state changes over time are typically classified as either *continuous* (and are referred to as *Continuous Variable Dynamic Systems*, CVDS) or *discrete*[2] (or *Discrete Event Dynamic Systems*, DEDS).

In CVDS, state changes occur continuously over time. Such systems are typically modelled by a set of ordinary/partial differential equations. Continuous

[1] *State* is the collection of variables that contain all the information necessary to describe the system at any point in time. Thus, state is basically a function of time [Bank86].

[2] Typically physical systems encountered in nature are treated as continuous while systems that are the result of human activity are treated as discrete; the latter includes the vast majority of systems that modern society is concerned with [Ho89].

systems are not pertinent to this thesis and are not discussed further; in his excellent classic book, Dugas [Duga35] chronicles the development of continuous dynamic systems from Aristotle to Kepler to Newton and to quantum mechanics.

In contrast to CVDS, in discrete systems, state changes (also referred to as *state transitions*) are assumed to take place only at a set of *discrete* instants in time rather than continuously. The happening whose occurrence causes a state transition to occur is referred to as an *event*. The discrete instants in time when state transitions occur are referred to as *event times*; the system state between two event times is called a *snapshot*.

The behaviour of Discrete Event Dynamic Systems cannot easily be described by ordinary or partial differential equations[3] [Ho89]. Several mathematical notations and techniques have been developed to allow the mathematical modelling of DEDS; these include Markov processes [Dynk82], Petri nets [Pete91], Queueing theory [Bund86] and Finite State Machines [Gill62].

Once a mathematical model has been constructed, it may be solved to provide answers to questions regarding the behaviour of the modelled system. This may be achieved analytically, by employing mathematical theory and deductive reasoning. Analytical approaches provide general solutions, however in practice only certain forms of equations may be solved. In this case the application of numerical methods is the alternative option for the solution of the mathematical model. Numerical methods employ computational procedures to produce solutions in steps: each step gives the solution for one set of conditions and the calculation must be repeated to expand the range of the solution [Neel87]. This stepwise philosophy makes numerical models particularly suitable for execution on computers.

[3] In principle, continuous modelling techniques may be applied to produce continuous approximations of discrete systems, as well as discrete approaches can produce discrete approximations of continuous systems [Neel87].

The process of executing a model on a computer system in order to derive answers to questions regarding the operation of the modelled system is referred to as *computer simulation.* A model adapted for simulation on a computer is known as a *computer simulation model*, or simply *simulation model.*

3.2 Discrete Event Simulation Modelling

Mathematical modelling may provide answers regarding certain characteristics of a discrete system, however it is often unable to capture the dynamic behaviour and other important aspects of the operation of the system in sufficient detail. Furthermore, in most real discrete event systems (e.g. stochastic problems), mathematical models have no simple and practical analytical or numerical solutions [Thes90] [Cars92] [Prit91].

In these cases, *discrete event simulation* (DES) is the only alternative available. In discrete event simulation, the computer program (the *simulator*) is not intended to solve mathematical equations, but rather encapsulates a detailed description of the structure and operational rules of the system under consideration; when executing, the program imitates the system, mimicking its operation.

Typically, a discrete event simulation model designed to run on a conventional von Neumann computer utilizes three main structures:

- A *global clock variable*, which keeps track of the progress of the simulation in terms of *logical or simulated time.*
- The *state variables*, which define the state of the system at any particular point in simulated time.
- The *event list*, which contains all events which have been scheduled but have not yet occurred (i.e. have not yet taken effect); a simulation event

may be viewed as modelling an event in the physical system which causes a state transition to take place.

Each scheduled event is assigned a *timestamp*, which denotes the point in simulated time that the event occurs. The simulator is driven by a main loop (referred to as the *simulation engine*), which repeatedly selects from the event list the event with the smallest timestamp, processes it and removes it from the event list. The processing of an event involves the execution of a piece of code, which describes the steps required for the appropriate state transition to occur, and the generation and scheduling of a number of new events into the simulated future [Bank86].

Discrete event simulation has several advantages and benefits over mathematical modelling or prototyping [Thes90] [Prit91] [Shan92] [Cars92]:

- It makes possible the testing and evaluation of systems, in cases in which the system does not exist, mathematical modelling is impossible and prototypes are expensive, time consuming or hazardous to build; it is generally easier, faster and cheaper to implement a simulator than building a prototype.

- It provides a higher degree of flexibility than prototypes as it can easily be modified. Thus it makes possible the efficient experimentation with new situations providing answers to "what if" questions which would otherwise be too costly and time consuming to contemplate. Consequently, it can reduce dramatically both the cost and duration of a system's development phase.

- It allows the representation of systems at any level of detail sufficient to meet the objectives of the designer, thus providing support to hierarchical design approaches.

- It provides for the study of the dynamic behaviour of systems by allowing the manipulation of time. Time can be compressed or expanded, thus providing a rapid view at long time horizons in the past or future of the system under consideration.
- The process of building a discrete event simulation model calls for the provision of a detailed description of the system, an activity which in itself may prove invaluable to enchance the designer's understanding of how the system operates.

Discrete event simulation modelling is an essential tool in the design, development and analysis of discrete event systems and is currently undergoing a dramatic increase in its range of application and an explosion of innovation [Ruta91] [Radi92]; innovation includes the exploitation of advanced software engineering [Fish92] and object oriented approaches [Zeig90] [Rumb91], use of Artificial Intelligence techniques [Roth89], the exploitation of different conceptual modelling frameworks [Zeig76] [Kreu86] [Derr89], and the development of specialized simulation languages and interactive support environments [SCS85] [Bart89] [Schr92] [Balm90] [Gord90] [Bank92].

3.3 The Need for Parallel Discrete Event Simulation

Parallel processing technology and the availability of extremely cost effective multiprocessor machines has had a significant and dramatic impact in the area of computer simulation.

The application of simulation to ever more complex discrete event problems has long placed it in the highly computation intensive world. Applications in which the computational requirements of simulations far exceed the capabilities of conventional sequential von Neumann computer systems include health care systems [Lomo88], training [Stei91], military systems [Wiel89] [Mors90], environment systems [Ebli89], [Pres90] [Bagr90] [Nico90], flexible manufacturing systems [Nevi90], automobile traffic modelling [Merr90], aerodynamic simulation [Norr87], telecommunication networks [Muft90] [Robe92] [Eick91] [Turn92], queueing networks [Reed88a] [Fuji89] [Fuji89a] [Chan89] [Ayan89] [Nico93] [Nico93] and Petri nets [Kuma90] [Thom91] [Nico91] [Bacc93] [Fers93].

The execution of these computationally intensive simulations requires either modelling at higher levels of abstraction in order to reduce the computational load[4] or more powerful machines[5]. The former is not considered a satisfactory approach as it does not allow the required detail to be incorporated in the model and may thus result to over-simplification of the investigated problem [Unge89]. Therefore, in the last fifteen years, the attention of modellers and simulationists has been directed to parallel, multiprocessor machines, as they see in them the potential solution to the demand for increased computational power.

3.3.1 Exploiting Parallelism

Three main approaches for exploiting parallelism at different levels in simulation problems have been developed and followed[6]:

- **Application Level.** This approach assigns independent replications of the same sequential simulation model (with possibly different input parameters)

[4]For the simulation of communications systems for instance, processing of an average of 5 million events per minute is typically required [Flec92]; higher abstraction levels aim to reduce the number of events that need to be processed.

[5]A third alternative is to apply statistical methods in order to reduce the number of runs required to make decisions regarding certain characteristics of the simulated system [Ripl87]; these techniques however are at still at an early stage of development [Righ89].

[6]Righter [Righ89] mentions one more approach, namely applying a parallelizing compiler to a sequential simulation program. Although this technique is transparent to the user, it exploits only a small portion of the available parallelism and thus is very rarely used.

onto different processors[7] [Bile85] [Heid86]. This approach can allow the exploration of large search spaces without the cost that parallelization of the code would involve (see subsequent sections). As no coordination is required between processors during the execution, unlimited scalability is possible. However, distribution of the entire simulation may not always be possible, due to memory limitations in the individual processors. Furthermore, this approach is not suitable for design environments wherein experiments must be sequenced, as results of one experiment are used to determine the experiment that should be performed next.

- **Subroutine Level.** Following this approach, different dedicated functional units (simulation engine subtasks or subroutines) are used to implement specific functions of the sequential simulation (e.g. random number generation, event list manipulation, state update, statistics collection etc); these units are then distributed to different processors. Examples of subroutine level parallel simulation may be found in [Wyat83] [Wyat84] [Comf84] [Kris85] [Wyat85] [Rees85] [Shep88] [Comf88] [Davi88]. The advantage of this approach is that it is transparent to the user. However due to the small number of subtasks in the simulation engine, only a limited amount of speedup may be achieved with a subroutine level distribution.

- **Event Level.** Neither of the two distribution levels above makes any attempt to exploit the parallelism of the physical system being modelled. In a system with inherent parallelism (and this includes the majority of real life systems), several state changes occur concurrently; the objective of event level distribution is the concurrent execution of the corresponding events. Two main techniques for the distribution of events may be distinguished based on the manipulation of the event list:

[7] Fujimoto uses the name *replicated trials* for this approach [Fuji90].

1. **Centralized.** In this technique, the event list is a centralized data structure maintained by a master processor. The simulation model is decomposed into functional modules which are then assigned to a set of slave processors. The master process selects from the event list the events that can be processed concurrently and distributes them for execution to the slave processors; new events scheduled by this execution are inserted back into the centralized event list [Jone86]. For the efficient implementation of this technique, shared memory multiprocessors are particularly suitable, with the event list being implemented as a shared data structure accessed by all processors [Jone89].

2. **Decentralized.** This technique involves the distribution of the event list onto the multiple processors, in order to allow the concurrent execution of events at different points of the simulated time. Decentralized, event level distribution is the simulation approach with the greatest potential for high performance and, consequently, has attracted considerable attention from the research community and has almost exclusively been employed for practical simulation applications. However, the distribution of the event list introduces synchronization problems which have been addressed by a number of researchers; these problems as well as a brief description of the most influential of the techniques that have been developed to address them are the subjects of the subsequent sections.

3.4 The Logical Process Paradigm

Typically, a physical system being modelled is viewed as consisting of a number of independent, concurrent, interacting, functional entities, which are referred to as *physical processes* (*PP*).

Decentralized parallel simulation strategies seek to divide the simulation model into a network of concurrent *logical processes* (*LP*), topologically identical to the physical system, with each logical process corresponding to a functional entity of the physical system (i.e. a physical process); this approach is referred to as the *Logical Process Paradigm*.

Interactions between physical processes are modelled by timestamped messages exchanged between the logical processes; the timestamp of each message denotes the point in time when the corresponding event occurs in the physical process being modelled by the receiving logical process. A logical process will repeatedly consume and process messages arriving on its input links possibly generating, as a result, a number of messages on its output links; a new input message will be accepted only after the processing of the preceding message has completed. For the calculation of the timestamp of an output message, the timestamp, as well as the simulated time required for the processing of its parent input message are taken into account.

Since each message corresponds to an event, each logical process may be viewed as possessing and processing a portion of the event list of the simulation model. The distribution of the logical processes onto the different processors of a multiprocessor machine enables the concurrent execution of events, thus providing for the exploitation of the parallelism in the physical system.

3.4.1 Timing Issues

The Logical Process Paradigm allows the *spatial* characteristics of the physical system to map naturally onto the simulation model. However, this does not hold for the *temporal* characteristics of the system, whose mapping onto the simulation model is not straightforward.

All physical systems obey the *causality principle* which defines the relationships between the various system states. More specifically, the causality principle requires that *the cause must always precede the effect in time*: state transitions which have some effect on some other transitions must occur before the latter, while state transitions that do not affect each other may take place in any order.

Thus, the causality principle imposes a partial ordering on the system's state transitions. This ordering of state transitions in the physical system also imposes an equivalent partial ordering on the corresponding events in the simulation model. In order to ensure that the simulation model faithfully and accurately reproduces the behaviour of the simulated physical system, the order in which logical processes receive, process and generate events must be the same as the order of the corresponding state transitions in the physical system.

In the physical system, causality is tracked using the physical time (i.e. a real time clock). In a sequential simulation model, wherein physical time is modelled by a single global variable, causality is preserved through adherence to the rule that events are processed in non-decreasing timestamp order [Fuji88]; in a distributed setting however, in which concurrent processing of events is allowed, preservation of this fundamental monotonicity property associated with causality is not straightforward.

The concept of causality between events is one of the fundamental problems in distributed computation[8] and has been addressed by a number of researchers including Lamport [Lamp78], Schneider [Schn82], Reed [Reed83], Gusella et al.

[8]Actually, causality has been a fundamental issue in both Greek/western (e.g. [Aris] [Aris-1]) and eastern (e.g. [Kalu75]) thought; in his "Laws", Plato dealing with causality says: "*When we find one thing changing another, and this in turn another and so on, of these things shall we ever find one that is the prime cause of change? How will a thing that is moved by another ever be itself the first of the things that cause change? It is impossible. But when a thing that has moved itself and that other a third and the motion thus spreads progressively through thousands upon thousands of things, will the primary source of all their motions be anything else than the movement of that which has moved itself?...It has been proved most sufficiently that soul is of all things the oldest since it is the first principle of motion.*" [Plat].

[Guse84], Chandy et al. [Chan85], Christian [Chri89], Dunigan [Duni91], Fidge [Fidg88] [Fidg91], Basten et al. [Bast94], Rabin [Rabi94] [Rabi94a] and Mizuno et al. [Mizu95].

With regard to distributed simulation, two main approaches have been developed to address the causality problem, usually referred to as *time driven* and *event driven* respectively; a further classification applies to these two main approaches which characterizes a technique as either *synchronous* or *asynchronous*[9]. Before the time driven and event driven approaches are discussed, a short description of the synchronous and asynchronous implementations is provided.

3.5 Synchronous versus Asynchronous Simulation

The synchronous framework requires that processes advance together in a lock step fashion. This is achieved by means of a *global* clock whose role is to synchronize the logical processes of the simulation model. The global clock may be implemented either using a centralized approach, with a single dedicated process to act as a synchronizer [Venk86] [Chri82], or in a distributed, decentralized fashion, where a more complex structure of dedicated processes is responsible for synchronizing the logical processes of the model [Peac79] [Baik85] [Conc89] [Zhan89].

The asynchronous approach allows the processes of the simulation model to operate asynchronously, advancing at completely different rates. Each process maintains a local clock variable which contains the current value of the simulated time[10]. This value represents the process's local view of the global simulated

[9] *Synchronous* and *asynchronous* are sometimes referred to as *tight* and *loose* respectively [Peac79].

[10] In distributed simulations, the domain of the local clock variables is the set of non-negative integers (i.e. scalar). Other schemes have also been proposed for asynchronous distributed

time and denotes how far in the simulated time the corresponding process has progressed.

One emerging theme in parallel simulation research is the study of techniques which combine aspects of both synchronous and asynchronous operation; these are usually referred to as *time window techniques*. These approaches involve barrier synchronizations to constrain asynchronous operation to be within some window of global simulated time; once the local clocks of all processes reach the end of the window, a global synchronization scheme is applied to allow processes to move to the next window. Usually, the window size is application dependent.

3.6 Time Driven Logical Process Simulation

In time driven simulation, the clock is an autonomous entity capable of incrementing *per se* and is the driving force of the simulation. The simulated time advances in time increments of fixed constant size (ticks or time steps). Each logical process must process all events in a particular time step before it is allowed to proceed with the events of the next time step.

In synchronous time driven simulation [Gilm86], *all* activity in the current time step must cease before the processes of the model are allowed to advance to the next time step. The asynchronous approach permits processes to begin executing events in the next time step as soon as their source processes (i.e. their predecessors) have finished the last time step [DeBe88]; the local clocks are synchronized by sending extra messages from each process to its successors.

Time driven simulation guarantees that the causality principle is not violated. However, its potential for speedup is limited to situations where the number of events to be processed by each and every process per time step is high;

systems whereby timestamps have the form of vectors (e.g. [Fidg88] [Fidg91] [Schm88] [Matt88]) or matrices (e.g. [Sari87]) [Rayn95].

otherwise starvation phenomena will occur as processes remain idle waiting for the clock to advance to the next time step. This approach is more appropriate for shared memory machines, as global synchronization is at present neither easily implementable nor efficient on asynchronous distributed memory machines [Fuji88].

3.7 Event Driven Logical Process Simulation

In the event driven approach, the driving force of the simulation which triggers actions is the availability of events to be processed. The simulated clock is a slave object which is incremented each time an event is executed to accommodate the duration of that execution.

Simulation may be either synchronous or asynchronous; event driven, time window techniques have also been developed including Moving Time Window (MTW) [Soko88] [Soko89] [Soko91], Bounded Lag [Luba88] and Conservative Time Windows (CTW) [Ayan92].

Like their time driven counterparts, synchronous event driven approaches keep the processes of the model synchronized using a global synchronization scheme, but with each update, the global clock is set to the minimum time of the next event for all processes, rather than the next clock tick [Peac79] [Baik85] [Conc89] [Zhan89].

In asynchronous event driven simulation, all computation in a logical process is initiated by the presence of messages on the process's input links. Upon receipt of an input message, the process will be activated to act upon the message and, as a result, update its local clock, which is set to the minimum next event time for that process. Processes are allowed to consume and execute messages as soon as they become available, without having to wait for a global clock to tick through periods of inactivity or for other slower, but unrelated, processes to advance. Events may be simulated simultaneously, even if they occur at completely different simulated times. Consequently, asynchronous event driven simulation has greater potential for high performance.

However, the concurrent simulation of events at different simulated times makes the model susceptible to violations of the causality principle. It has been shown (e.g. Lamport [Lamp78] and Misra [Misr86]) that a distributed system consisting of logical processes which operate asynchronously and interact exclusively via timestamped messages, will adhere to the partial ordering imposed by causality constraints in the physical system, if each logical process consumes and processes events in non-decreasing timestamp order; this condition is referred to as the *local causality constraint* [Fuji88].

Thus, the problem of guaranteeing that the model implements exactly the same global causal precedence relationships of the physical system, reduces to ensuring that each logical process obeys the local causality constraint and processes messages in non-decreasing timestamp order.

If each process in the distributed model has a single input link (i.e. it receives messages from just one source process), then adherence to the local causality constraint is straightforward. Indeed, in single input processes, there is a direct correlation between order of arrival and order of consumption. Assuming that the timestamps in the input link are ordered in time then the output timestamps are also guaranteed to be ordered. Thus on any particular output (and therefore input) link, messages will be issued in non-decreasing timestamp order.

However, most, if not all, practical simulation models will include multiple input processes which receive messages from more than one source process; such processes are usually referred to as *merge* processes. In this case, the order in which messages arrive on the different input links does not adhere strictly to the order in which the corresponding events occur in the physical system; it merely depends on the relative real time propagation delays of the messages in the distributed machine and not on the simulated time timestamps that these messages carry.

Therefore, in the general case, messages will arrive at the merge processes of the model in a non-increasing timestamp order. Consequently, immediate consumption and processing of messages by merge processes may result in violation of the local causality constraint; the processing of an out of order message (i.e. a message which according to its timestamp should have been processed in the simulated past) is referred to as a *preemption*.

Several techniques have been developed to address the preemption problem in asynchronous event driven simulations and to ensure that the local causality constraint is not violated. These techniques employ different synchronization protocols to enable processes to decide when it is safe to consume and process events.

Asynchronous protocols are traditionally classified into two broad categories, namely *conservative* and *optimistic*[11]; hybrid approaches with both, conservative and optimistic behaviour also exist [Luba89a] [Dick90] . Detailed surveys of the various asynchronous event driven simulation techniques developed to date may be found in [Righ89] [Fuji90] [Fuji92] [Fuji93] [Fers94] and [Nico94]; the following sections provide a short presentation of the most influential of them.

3.7.1 Conservative Techniques

Conservative techniques were originally proposed by Chandy and Misra [Chan79a] and Bryant[12] [Brya77]. These techniques allow a logical process to accept and process an event only if it is absolutely safe to do so, thus strictly avoiding the

[11]Reynolds has introduced an alternative, more detailed taxonomy [Reyn88] [Reyn89]. However, within the scope of this thesis, the conventional taxonomy that distinguishes between conservative and optimistic techniques is sufficient.

[12]Conservative techniques are often referred to as the Chandy-Misra-Bryant (CMB) protocols.

possibility of a preemption ever occurring.

In order to determine when it is safe to process an event, it is required that messages from any process to any other process be transmitted in chronological order according to their timestamps. This ensures that the timestamp of the last message issued on an input link provides a lower bound on the timestamp of any subsequent message that will later arrive on the same link; this bound (i.e. the timestamp of the current message pending processing on an input link or, if no such message exists, the timestamp of the last message received on that link), is referred to as the *link clock*. A merge process repeatedly selects the input link with the smallest clock and, if there is a message pending, it is consumed and processed; otherwise the process is forced to block until a message is issued on the link, as the timestamp of that message might be less than the timestamps of the messages currently waiting on the other incoming links.

This algorithm guarantees that merge processes will process messages in increasing timestamp order, thus ensuring adherence to the local causality constraint. However, deadlocks may occur if a message expected by a blocked process is not eventually issued; this situation may occur if a cycle of processes is formed, with each process being blocked due to its waiting for a message from another process in the cycle [Peac79] [Righ89] [Fuji90].

3.7.1.1 Deadlock Avoidance

In order to address the deadlock problem, Chandy and Misra [Chan79] [Chan79b] have proposed a *deadlock avoidance* mechanism, whereby *Null messages* are used continuously to advance the output link clocks, even if no events are issued on the link; Null messages inform the corresponding receiving process of the minimum potential timestamp of the next event to appear on the link thus preventing it from indefinitely blocking upon that link. This scheme is based on the assumption that each process has the ability to predict its actions in the immediate simulated future so that it may calculate a lower bound on the next outgoing message on each of its output links. Each time a logical process finishes processing an event, it issues to its output links a Null message with that bound; based on this information the receiving process calculates a new bound for its output links and so on. The larger the output timestamp bounds that may be predicted by a process, the smaller the number of Null messages generated and the less the time that a receiving process blocks upon its input links.

The ability of a process to predict its simulated future is referred to as *lookahead*. The most common type of lookahead is the *Minimum Timestamp Increment*, which is the minimum processing simulated time for each message passing through the process [Fuji90]; another form of lookahead is the *distance* between two causally related but not necessarily directly connected processes, namely a lower bound in the amount of simulated time that must elapse for an unprocessed event to propagate from one process to the other [Ayan89]. Lookahead is an important aspect of conservative simulation and is essential for both the correctness and performance of protocols and several mechanisms have been developed which attempt to exploit the characteristics of the simulated system to improve lookahead [Nico88] [Lin89] [Wagn89] [Louc90] [Pete93]. Generally, the lookahead information required for the calculation of timestamps of the Null messages is application dependent and is explicitly provided to the processes in advance by the simulation programmer, although attempts for their automatic calculation have been made [Cota90].

Null messages are used only for synchronization and impose a significant communication overhead which may dramatically affect the performance of the simulation [Fuji90]. To address this problem, a number of optimizations have been proposed which aim to reduce the number of Null messages generated [Holm78]

[Nico84] [Misr86] [Bain88] [Su89] [DeVr90]; the *carrier Null message* protocol [Cai90] [Wood94] attempts to reduce the number of Null messages by using them to acquire/propagate additional knowledge to the processes in order to enhance their lookahead. Another approach is to eliminate the need for Null messages by exploiting the structural characteristics of the application in order to eliminate the cycles in the model which may cause deadlocks [Lin90].

3.7.1.2 Deadlock Detection and Recovery

Another approach to deal with the deadlock problem is the *deadlock detection and recovery* scheme proposed by Chandy and Misra [Chan81], whereby the simulation proceeds until it deadlocks (the *parallel phase*), and when deadlock is detected, it is resolved (the *synchronization or interface phase*) to enable the next parallel phase to commence. Distributed deadlock detection algorithms have been proposed by Dijkstra and Scholten [Dijk80], Misra [Misr86] and Groselj and Tropper [Gros89]. Various other deadlock detection algorithms have been proposed which address particular situations: Kumar and Harous [Kuma91] propose deadlock detection algorithms for simulation of queueing networks, Reed and Malony [Reed88b] and Fujimoto [Fuji89] have investigated the deadlock detection problem in shared memory machines, Misra [Misr86] has proposed alternative algorithms for detecting deadlocks when only a portion of the process network has deadlock, Groselj and Tropper [Gros88] propose a scheme to be used for deadlock detection within one processor, and Liu and Tropper [Liu90] propose algorithms that target specific types of cycles of blocked processes.

3.7.1.3 Characteristics of Conservative Protocols

The major advantage of the conservative techniques is that they are simple and straightforward to implement. However, the requirement for strict adherence to the local causality constraint does not allow full exploitation of the inherent parallelism of the model. Conservative techniques require static configuration of the distributed model [Fuji90]; systems with dynamic behaviour generally can not be easily modelled. Furthermore, conservative protocols rely heavily on the lookahead, and are thus suitable only for applications with good lookahead properties. The knowledge required for the exploitation of lookahead must be explicitly provided by the simulation programmer. This implies that in order to design a correct and fast synchronization protocol, the programmer must be familiar with the details of both, the simulated system and the synchronization protocol; this requirement is considered the most serious drawback of conservative techniques although research into adding transparency to conservative algorithms is continuing [Jha93].

With regard to performance of conservative techniques, reported speedups include 16 on 25 processors of a Sequent Balance and 1900 on a 16384 processor Connection Machine [Luba89] [Luba89b], 7 on 12 and 9 on 24 iPSC processors [Chan89], 18 on 31 transputers [Merr90], 8 on 64 and 10 to 20 on 128 iPSC processors [Su89] and 16 on a 16 processor BBN Butterfly [Fuji89].

3.8 Optimistic Synchronization Protocols

The objective of optimistic approaches is to detect and recover from causality errors rather than strictly avoid them. The most important and influential optimistic mechanism is *Time Warp* (or *Virtual Time*[13]) proposed by Jefferson and Sowizral [Jeff82] [Jeff85] [Jeff85a].

[13]Jefferson proposed *Virtual Time* as a paradigm and *Time Warp* as a mechanism to implement it, but the two terms are used interchangeably.

3.8.1 Time Warp

In Time Warp, processes consume and process events as they arrive, without first deciding whether such an action is safe or not. When a preemption is detected (i.e. when an event arrives whose timestamp indicates that it should have been processed in the past - this event is called a *strangler* in Time Warp), the process "rolls back" in simulated time and undoes all its actions up to the point indicated by the timestamp of the strangler. It also sends *anti-messages* to successor processes to inform them of the preemption and cancel the effects of previous erroneous messages issued by the process; upon receipt of an anti-message, successor processes follow the same procedure, undoing their actions and generating more anti-messages.

Anti-messages may be sent immediately upon detection of a preemption (*aggressive cancellation*) or at a later time, after the process has established that the re-execution of its actions generate different output messages (*lazy cancellation)* or different process states (*lazy re-evaluation*). Lazy cancellation avoids unnecessary rollbacks and thus improves performance, however, if rollbacks are necessary, aggressive cancellation will enable them to occur sooner, thus preventing erroneous computation from spreading further into the model; it has been shown that in most cases lazy cancellation tends to perform better that its aggressive counterpart [Gafn85] [Gafn88] [Lomo88] [Righ89].

3.8.1.1 Global Virtual Time

In order to have the ability to cancel past actions, each process maintains a record of its past history which is periodically updated; typically, this record includes past state vectors, processed input events, and previously sent output messages. For the rollback of the simulation to be feasible, each process must hold information regarding its history up to last "correct time", which generally is the smallest local clock value amongst all independent processes. This value is referred to as the *Global Virtual Time* (GVT); algorithms for the computation of the GVT have also been developed [Sama85] [Bell90] [Lin90a] [Prei89].

3.8.1.2 State Saving and Memory Management

The periodic need to save and maintain information regarding the past history of the logical processes has a significant impact on both the performance and the memory requirements of the simulation [Fuji90] [Akyi93] [Prei92] [Bell92] [Clea94].

State saving may require the traversal of complex dynamic data structures, a function which involves a significant amount of computation and can degrade the performance of the simulation, even if the state vector is of relatively moderate size. To address this issue, the use of hardware support for speeding up state saving has been proposed [Fuji89b] [Fuji88a]. This approach however does not provide a solution to the demand for large amounts of storage to maintain process history records.

Infrequent state saving can reduce both the time spent during the saving and the amount of information needed to be stored but increases the cost of rollbacks, as processes have to roll back further into the simulated past, thus cancelling and reexecuting more events [Lave83] [Mitr84] [Lin90b].

Another approach to reduce the memory requirements is the frequent computation of GVT; each time the GVT is computed, storage used for history before the new GVT may be reclaimed (this is referred to as *fossil collection* [Jeff85a]). However, the computation of GVT involves global synchronization and, consequently, frequent update of GVT can degrade performance.

The aforementioned techniques attempt to manage memory so that the simulation does not run out of space; a number of techniques have also been developed to address the problem of freeing memory when this situation does indeed occur. These techniques include *message sendback* [Jeff85a] [Gafn88], *cancellback* [Jeff90] and *artificial rollback* [Lin92]; the basic idea behind these mechanisms is to rollback overly optimistic computations and reclaim the memory they use.

3.8.1.3 Characteristics of Optimistic Protocols

Optimistic synchronization approaches can accommodate dynamic creation of logical processes [Tink89], and are typically more transparent to the simulation programmer. However they are more complex to implement due to the requirements for effective memory management and are more difficult to debug than their conservative counterparts [Fuji90]. Reported speedups achieved by optimistic techniques include 10 to 20 on a 32 processor hypercube machine and 27 using 100 processors of a BBN Butterfly [Fuji90] and 57 on a 64 node hypercube [Fuji89a].

3.9 Modelling and Simulation in Computer Architecture Research

Technological and architectural advances have dramatically increased the size and complexity of computer system designs. The need to cope with this complexity and the requirement for shorter development times and reduced cost have assigned key roles to modelling and simulation in computer architecture research. Modelling and simulation are essential tools for experimenting with alternative ways of using the available silicon area, verifying the timing behaviour and functional correctness and measuring the performance of alternative architectural designs.

A digital system may be modelled at different levels of abstraction (figure 3.2) [Hart87] [Arms89]:

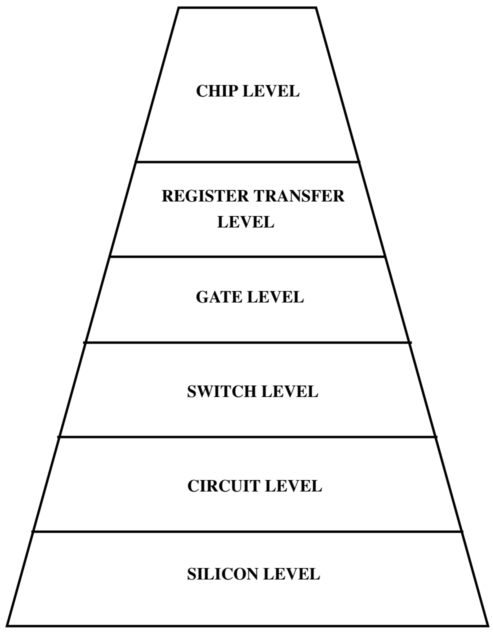


Figure 3.2: Abstraction Levels in Digital Systems

- *Silicon Level.* At this level the real geometry of the physical layout of materials such as diffusion, polysilicon, and metals on the silicon surface is described. Typically the layout is specified by means of a schematic, graphical representation, though textual layout formats are also used (e.g. the "Caltech Intermediate Form" [Mead80]). The layout is a symbolic, non-mathematical model, a purely morphological description, without any behavioural information incorporated and therefore can not be used directly for simulation.

- *Circuit Level.* At this level the system is expressed in terms of traditional passive and active electrical circuit components such as resistors, capacitors and bipolar or MOS/CMOS transistors. The circuit may be specified as a schematic diagram or via a textual description; the textual version of a circuit diagram is called a circuit level *net list.* The circuit net list may be specified explicitly or be extracted from the physical layout or circuit schematics. At circuit level, the input and output of each component in the model have analogue values. The analogue behavior of the components at this level is typically expressed in terms of differential equations which may be solved by a circuit simulator (e.g. SPICE [Nage73]); circuit simulators typically use as input the circuit net list.

- *Switch Level*. At this level, the system is described at the same level of abstraction as in the circuit level, however only digital, rather than analogue, signal values are considered (i.e. {1,0}, {on, off}, {high, low}). The same circuit diagram and circuit net list are used, but components are modelled in a much simpler way, wherein, instead of a precise analogue behaviour only the digital behaviour is described. Transistors are modelled as switches, with two states, namely "on" (low impedance) and "off" (high impedance). A model at switch level, and all levels above that, may be simulated using a discrete event simulator.

- *Gate Level*. This is the logic design level, where the implementation of the system in terms of gates (AND, OR, inverters etc) and flipflops is described. This description may be provided graphically, by means of a logic diagram, or textually, using a *Hardware Description Language* (HDL); a gate level net list may be automatically extracted from the schematic or textual specification. The net list may be provided as input to a discrete event logic simulator. Gate level has traditionally been the main design level for digital systems.

- *Register Transfer Level (RTL)*. Here, the model of the system is expressed in terms of higher level components such as registers, counters, multiplexers, ALUs, multipliers, shifters, memory blocks etc.; these components are sometimes referred to as *functional blocks*, assigning the term *functional level* *modelling* to RTL descriptions. Although, in principle, RTL components may be expressed in terms of an interconnection of lower level primitives (such as gates) this is not the design approach adopted at this level; rather, RTL components are primitive behavioural models directly expressed using a functional HDL. RTL models may be simulated by a functional discrete event simulator.

- *Chip (or Architectural) Level.* At this level, the structural primitives of the model are blocks such as processors, memories, serial and parallel ports, interrupt controllers etc. Typically, the model boundaries are defined by the chip boundaries, however this requirement is not restrictive (e.g. when modelling parallel architectures, where the system consists of more that one chip). As with the functional blocks in the Register Transfer level, chip level components are primitive behavioural models and are not specified hierarchically in terms of lower level blocks. Discrete event simulation is employed for the execution of chip level models.

A number of Computer Aided Design tools have been developed and are commercially available [Wern84]. Typically, these tools allow the modelling of the system at different levels of abstraction (sometimes providing automatic transformation of the model from level to level), provide a Hardware Description Language for the behavioural specification of functional blocks and employ a discrete event simulator for the simulation of the modelled system.

3.9.1 The Need for Improved Digital System Simulation Performance

The level of abstraction at which a system is modelled has a direct impact on the performance of the simulation of the model. Switch and logic level models typically consist of hundreds of thousands of components, whose simulation on conventional von Neumann machines is extremely time consuming. The increasing complexity of architectural designs has also dramatically decreased the speed of higher level simulators (e.g. Register Transfer and chip level); this is particularly true in the design of parallel architectures, where the simulation of the parallel execution of even a small workload can consume large amounts of CPU time, while in order to obtain a reliable performance evaluation of the system, large workloads need to be simulated. Long execution times have made simulation a major and increasing bottleneck in the VLSI design process.

Various approaches have been followed to speed up the simulation of digital systems. One such approach makes use of special purpose hardware accelerators to execute the simulation; these can achieve impressive simulation performance, however, they are expensive and have limited flexibility in terms of simulated element types and delay models [Avra83] [Agra87] [Blan84]. Another technique is partial simulation, wherein only parts of the system are tested through simulation; this can accelerate the overall design process but entails the danger of overlooking costly design errors.

3.9.1.1 Parallel Digital System Simulation

An alternative approach to speed up simulation, is to employ parallel simulation techniques whereby gates, functional blocks, etc. are modelled as logical processes, and execute the simulation model on a multiprocessor machine. This approach has the potential for higher performance as digital systems typically have a degree of inherent parallelism, and consequently has received considerable attention and interest.

Su and Seitz [Su89] have used variations of the conservative deadlock avoidance algorithm (Null messages) for gate level simulations and have achieved a speedup of 8 on 64 and 10 to 20 on 128 processors of an Intel iPSC machine.

Soule and Gupta [Soul91] have examined gate level simulations using both, deadlock detection and recovery algorithms and centralized simulation on shared memory machines achieving a speedup of 16 to 29 on an 64-processor parallel machine, 6 to 9 on a 14-processor Encore Multimax and 2 to 4 on a 16-processor DASH respectively.

DeBenectitus et al. [DeBe91] examines gate level simulation, using a conservative algorithm which eliminates cycles in the logic diagram.

Briner [Brin91] and Sporrer and Bauer [Spor93] use Time Warp for switch and gate level simulations on a 32-processor BBN GP100 machine reporting speedups from 5 to 12 and 4 to 8 respectively.

Comparison studies of various synchronization protocols for gate level simulation have been performed by Lin et al. [Lin90c], Chung and Chung [Chun91] and Manjikian and Loucks [Manj93]; those report that optimistic protocols yield better performance.

Parallel techniques have also been applied for simulation at the architectural level (parallel architectural simulation).

Yu et al. [Yu89] use time driven algorithms to simulate multistage interconnection networks, reporting speedups ranging from 8 to 14 on a 16-processor Sequent machine; Ayani and Rajaei [Ayan90] also simulate multistage interconnection networks but using a conservative protocol.

Lin et al. [Lin89a], Heidelberger and Stone [Heid90] and Nicol et al. [Nico92] employ parallel techniques for the simulation of cache memories driven by memory reference traces.

Reinhardt et al. [Rein93] use synchronous conservative algorithms to simulate a shared memory parallel architecture while Bailey and Pagels [Bail91] employ the deadlock avoidance algorithm for the simulation of bus-based multiprocessors. Konas and Yew [Kona92] compare the performance of the deadlock avoidance algorithm with Time Warp and a synchronous technique based on a global clock for simulating a synchronous multiprocessor system; they conclude that the synchronous, global clock method yields the best performance.

3.10 Summary

This chapter has presented a short overview of modelling and simulation, with emphasis on distributed simulation. The causality issues arising in distributed simulations have been discussed and the various techniques that have been developed to address these issues have been presented. The chapter has concluded by discussing the role of modelling and simulation in digital system design.

The next two chapters deal with asynchronous hardware systems, and in particular with issues related to the modelling and simulation of such systems.

Chapter 4

Asynchronous Systems

4.1 Introduction

A digital system is typically designed as a collection of sub-systems, each performing a different computation and communicating with its peers to exchange information.

Before a communication transaction takes place, the sub-systems involved need to synchronize, namely to wait for a common control state to be reached, which guarantees the validity of data exchanged.

In synchronous systems, the synchronization of communicating sub-systems is achieved by means of a global clock whose transitions define the points in time when communication transactions can take place. The operation of a synchronous system proceeds in lockstep, with the different sub-systems being activated to perform their computations in a strict, predefined order.

Another digital design philosophy allows sub-systems to communicate only when it is necessary to exchange information. The operation of the system does not proceed in lockstep, but rather is *asynchronous*; each sub-system operates at its own rate synchronizing with its peers only when it needs to exchange information. This synchronization is not achieved by means of a global clock but rather, by the communication protocol employed.

Asynchronous design techniques have been explored since, at least, the mid 1950s by a number of researchers including Huffman [Huff54], Muller and Bartky [Mull56], Unger [Unge69], Miller [Mill65], Keller [Kell74] and Seitz [Seit70] [Seit80]. An influential contribution in the field was the Macromodules project at Washington University, St. Louis, where Molnar and Clark demonstrated the design simplicity and modularity resulting from asynchronous logic [Clar67] [Clar74]. Some of the early mainframe computers, such as MU-5 and Atlas at Manchester University [Ibbe78] [Lavi78], were constructed as entirely asynchronous systems.

Despite these efforts, the asynchronous approach has not hitherto been established as a major philosophy in digital design due to a number of inherent difficulties of asynchronous logic. The asynchronous, concurrent, operation of sub-systems significantly complicates the task of specifying the ordering of computations so that correct functionality is ensured; in synchronous systems the ordering of operations is fixed by the placement of latches and the global clock. Furthermore, in synchronous systems, circuit hazards and dynamic states may easily be dealt with by adjusting the clock period; addressing these problems in asynchronous systems is somewhat more complicated [Hauc93]. As a result, synchronous techniques have been favoured by the VLSI design community and most current digital design is based upon the synchronous approach.

However, recently, there has been an resurgence of interest in asynchronous design techniques worldwide, due to the several potential benefits that the elimination of global synchronization may offer.

4.2 Advantages of Asynchronous Systems

Four major areas in digital design may benefit by the application of asynchronous logic, namely clock distribution, power consumption, performance and technology migration.

4.2.1 Clock Distribution Problems

In a synchronous clocked system, the global clock signal must be distributed across the silicon to control the operation of the different circuit elements. However, varied propagation delays prevent the clock signal's arriving at all circuit elements at exactly the same time; the difference in arrival times of the clock signal at different parts of the circuit is referred to as *clock skew*. Clock skew is typically accommodated by longer clock periods, which imply a reduced maximum clock frequency. Advances in VLSI technology have led to a significant decrease in process feature size and an increase in the number of devices which can be built on a single chip. As VLSI systems become smaller, denser and faster, clock skew becomes increasingly severe and accounts for more of the design expense [Dobb92]. The lack of global synchronization in asynchronous systems eliminates concerns regarding clock skew.

4.2.2 Potential for Low Power

Power consumption is increasingly a major concern in the rapidly growing market for portable equipment, where battery life is crucial, and in the design of high performance RISC processors, where high heat dissipation introduces difficulties in packaging as well as cooling CMOS VLSI devices [Furb95]. In CMOS, the power dissipated is proportional to the clock frequency [Eshr89].

Lower voltage VLSI processes can reduce power consumption but not to a degree sufficient to meet the performance requirements of current and future VLSI systems[1] [Furb95]. Power consumption can be further reduced by activating only

[1]For example, a decrease in supply voltage from 5V to 3V reduces the power by a factor of 3 while a decrease to 2V reduces power by a factor of 6 [Furb95].

those units of the system that do useful work, i.e. those used in the current computation. Typically, in synchronous systems, the global clock toggles clock lines, charging and discharging capacitance throughout the silicon area, even in portions of the circuit unused in the current computation. To address this problem logic design techniques which gate off clock signals from certain areas of the circuit, and architectural techniques which attempt to eliminate redundant operations (such as speculative prefetching) have been developed [Furb95] [May94]. These approaches, however, compromise the flexibility of the design and increase its complexity, making global synchronization even more difficult.

In asynchronous systems, circuit components are activated only when necessary, while at other times they remain idle without dissipating significant power.

4.2.3 Potential for High Performance

As already mentioned, the elimination of clock skew removes the limitation imposed on performance by the need for extended clock periods and hence, reduced clock frequencies; furthermore, the lower power consumption promised by asynchronous logic may allow increased supply voltages with decreased heat dissipation and, consequently, increased performance. However, there is a more direct impact that asynchronous design may have on the performance. Synchronous systems are optimized for worst-case conditions; the clock period is adjusted according to the time that might be required for the slowest operation to complete, even though, in general, the average case will complete in a much shorter time (e.g. in a ripple-carry adder). Asynchronous systems can be optimized for the average-case conditions with each operation taking as long as required for any particular situation.

4.2.4 Better Technology Migration Potential

Asynchronous philosophies allow a system to be designed as a set of sub-systems communicating via well defined interfaces (see section 4.3.2). Since there is no global synchronization, components in an asynchronous system may be substituted by faster ones, without altering other components or the overall structure of the system, providing the interfaces are compatible; the substitution of a component may directly affect the performance of the overall system. In a synchronous system, the overall performance depends on the worst-case conditions and therefore, it is often the case that in order to exploit the speed potential offered by a new technology, reorganization of the whole system is required to deal with the new worst-case conditions.

4.3 Basic Characteristics of Asynchronous Systems

Current asynchronous designs are typically categorised by the timing model they assume, the signalling protocol they use and the technique they employ for the transfer of data between two elements.

4.3.1 Timing Model

The timing model defines the assumptions which are made regarding the circuit and signal delays of the system. Two main categories are typically distinguished:

- *Delay insensitive* systems guarantee correct functionality regardless of the delays in circuit elements and the delays in the wires which connect them. A *speed independent* system ignores delays in circuit elements but assumes zero delays in the wiring.

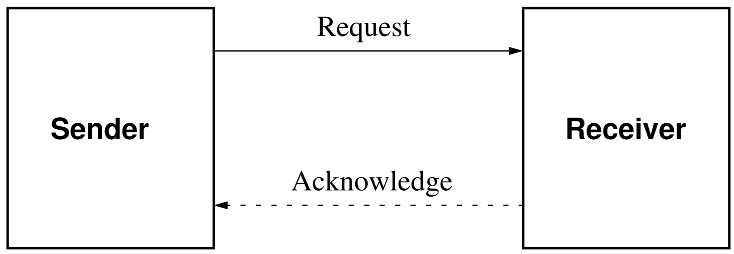


Figure 4.1: The Request-Acknowledge Interface

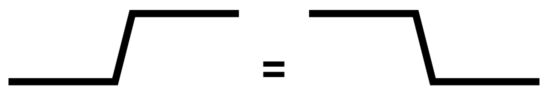


Figure 4.2: Two-phase Signalling: Rising and Falling Edges Equivalent.

- *Bounded delay* systems base their correct functioning on the assumption that processing and communication delays are bounded by some predefined upper limit.

4.3.2 Signalling Protocols

The signalling protocol specifies the sequence of events which must take place in a communication transaction between two elements of the asynchronous system. Typically, an asynchronous communication protocol employs two signals, namely a request and an acknowledge as depicted in figure 4.1. A communication transaction between a sending and a receiving element can be considered as having two or four phases.

4.3.2.1 Two-phase Signalling

Two-phase signalling recognizes and responds to transitions of the voltage on a wire, regardless of whether the transition is rising or falling; rising and falling edges are equivalent and carry the same information (figure 4.2). A transition is

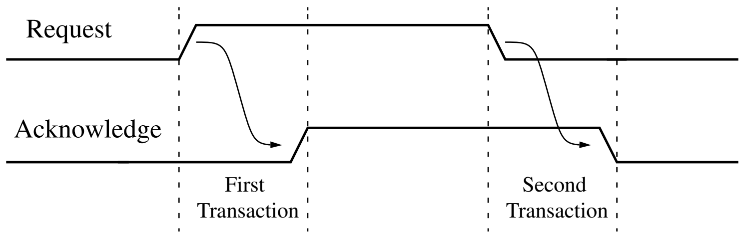


Figure 4.3: Two-phase Signalling Protocol

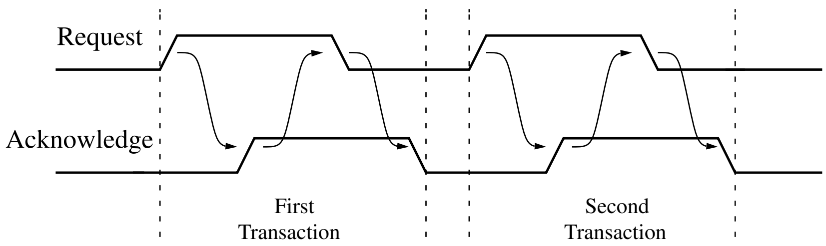


Figure 4.4: Four-phase Signalling Protocol

referred to as an *event*; two-phase signalling is also called *transition signalling*.

Figure 4.3 illustrates the interaction between a sending and a receiving element using two-phase signalling. The sender initiates the transaction and issues a request event to the receiver by causing a transition on the request wire (first phase); the receiver responds by issuing an event on the acknowledge wire (second phase). Rising and falling transitions on the request wire alternate, each transition initiating a new communication transaction.

4.3.2.2 Four-phase Signalling

In four-phase signalling only one type of transitions (typically rising) is used to signal events; the other, the falling, is used once the transaction is complete, to return wires to their initial state. Four-phase signalling is illustrated in figure 4.4. The first two phases of the transaction are similar to the transition signalling protocol (i.e. request *high* - acknowledge *high*) but in this case they are followed by another two phases which restore the wires to their initial state (i.e. request *low* - acknowledge *low*).

4.3.3 Data Passing Techniques

The request and acknowledge signals are used to regulate the flow of information between two communicating elements in the asynchronous systems. This information is a set of bits, with each bit being either "1" (high) or "0" (low). A variety of techniques have been developed to encode the value of each bit being transmitted during a communication transaction.

4.3.3.1 The Four-Wire Technique

The *four-wire* technique uses two pairs of request-acknowledge signals, one for each value of the transmitted bit. An event on one request signal denotes a "1" while an event on the other indicates a "0". The two request signals are mutually exclusive; the value of a bit cannot be both "0" and "1" at the same time. Furthermore, in every transaction there is always an event on one of the two request wires for each bit; thus, the entire data word has reached the receiver when an event has been detected for each bit of the word. A new transaction will not commence until an event has been detected by the sender on an acknowledge signal for each bit.

4.3.3.2 The Three-Wire Technique

The *three-wire* technique is similar to the four-wire mentioned above, but uses one acknowledge wire, instead of two, per pair of request wires. This scheme has the advantage of using fewer wires per bit of information.

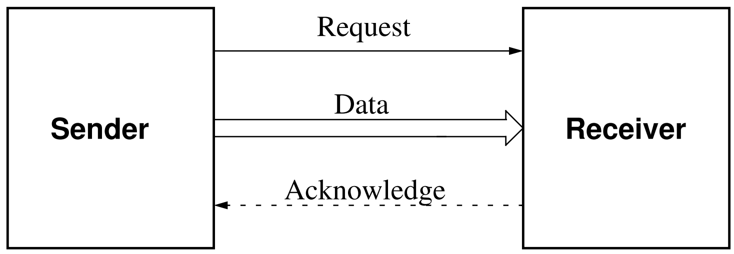


Figure 4.5: The Bundled Data Interface

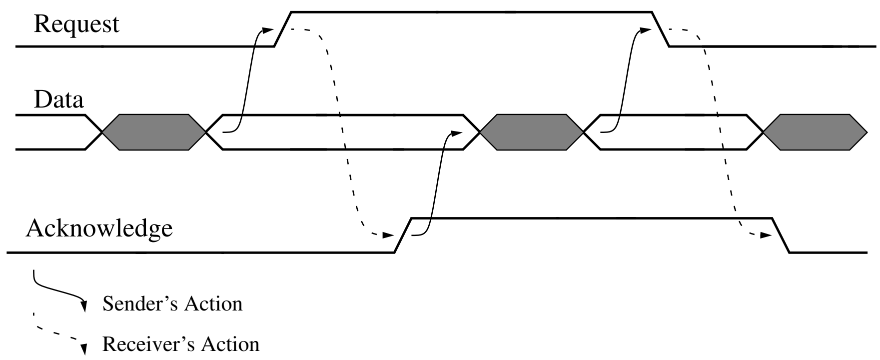


Figure 4.6: The Two-phase Bundled Data Protocol

4.3.3.3 The Two-Plus-Wire Technique

Another variation of the two aforementioned techniques is the *two-plus-wire* scheme, whereby two request wires per bit are used, one for encoding each of the two values, but the acknowledgements for all bits are combined into a single event which is transmitted over a single wire. Thus, for an n-bit data word, $2n+1$ wires are required. The two-plus-wire scheme, like both its aforementioned variants, may use transition or four-phase signalling for the communication of the request and acknowledge events; if four-phase signalling is employed the protocol is referred to as *dual rail encoding*.

4.3.3.4 The Bundled Data Technique

The *bundled data* technique employs a single pair of request and acknowledge signals for the entire data word. This scheme is illustrated in figure 4.5. As for the three techniques mentioned above, transition or four-phase signalling may be used. Figure 4.6 illustrates the two-phase bundled data protocol. The sender places the data to be transmitted on the data wires (grey area in figure 4.6) and then initiates a communication transaction by issuing a request event to the receiver. Upon detecting the request event, the receiver commences the processing of the data. When the processing is completed, the receiver issues an acknowledge event to the sender, whereupon the sender can remove the data and start preparing the next value; the data must be kept stable until the sender receives the acknowledgement from the receiver.

Contrary to the three data passing techniques described in sections 4.3.3.1-4.3.3.3, which are delay insensitive, the bundled data technique is based on the bounded delay model; all transitions on the data wires must be observed at the receiver before the request event, i.e. the delay on the data wires must be less than the delay on the request signal. This requirement is known as the *bundled data delay constraint*.

4.4 Micropipelines

In his influential 1988 Turing Award lecture, Ivan Sutherland introduced a new conceptual framework for designing asynchronous systems [Suth89]. Within this framework an asynchronous system is designed as a set of "Micropipelines". A Micropipeline is a simple, data processing, elastic[2] pipeline whose stages operate asynchronously and communicate using the two-phase bundled data protocol.

[2]A pipeline is *elastic* if the amount of data in it may vary in time. The input and output rates of an elastic pipeline may not be equal.

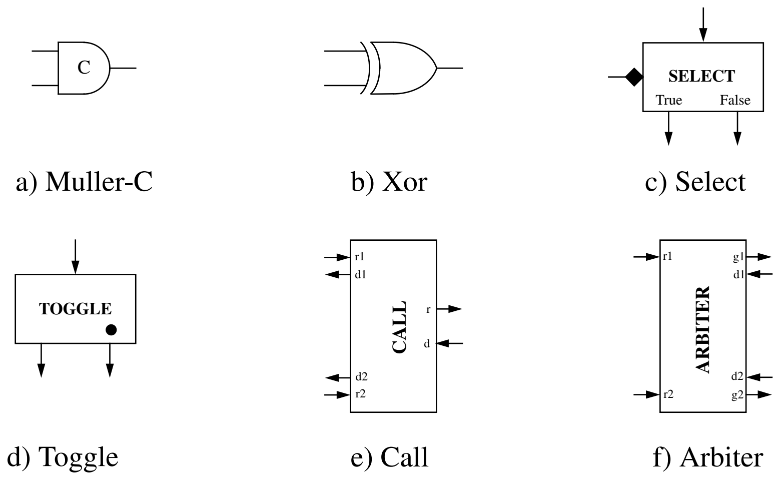


Figure 4.7: Event Control Modules

Sutherland also proposed a set of event control blocks for the design of control circuits in micropipelined systems as well as event controlled storage elements to be used in such systems.

4.4.1 Event Control Elements

The basic set of event control blocks proposed by Ivan Sutherland is depicted in figure 4.7:

- The *Muller-C* element provides the AND function for transition signals (events). An event is generated on the output side only if an event has been received on both inputs of the Muller-C.

- The *Xor* element implements the OR function for transition events. An event arriving on either input will cause an output event to occur; for correct operation, events should not be issued simultaneously on both inputs. Xor is also referred to as *merge*.

- The *Select* block implements a conditional test, steering events arriving on its input to the appropriate output according to a Boolean select signal (denoted by a diamond in figure 4.7c).

- The *Toggle* block also steers input events to one of its outputs, however this is not done on the basis of a Boolean test but alternately, starting with the output indicated by the bullet (figure 4.7d).

- The *Call* block is used to allow two separate circuit components to share access to a single sub-circuit. The two requesting components may issue request events on $r1$ and $r2$ respectively. Simultaneous events on both request inputs are not allowed; a new input request will be issued only after the processing of the previous input event has been completed. Upon receipt of an input request, the Call generates an output request (r); when the acknowledge signal is issued by the requested sub-circuit (d), Call steers it to the appropriate requesting component (either $d1$ or $d2$). The functionality provided by the Call element is analogous to a procedure call in software.

- The *Arbiter* block is used to permit two independent, asynchronous circuit components mutually exclusive access to a common sub-circuit. Request events may arrive (on $r1$, $r2$ wires) at arbitrary times and it is the responsibility of the arbiter to guarantee that only one request event is let through (issuing an event on either $g1$ or $g2$) and served at each particular moment. Typically the first request to arrive is granted service (in a fashion similar to the Call block) while the arbiter, acting like a semaphore, delays subsequent grants until after the acknowledge signal (d) corresponding to the previous request has been received. If both input requests arrive simultaneously, an arbitrary, non-deterministic choice is made.

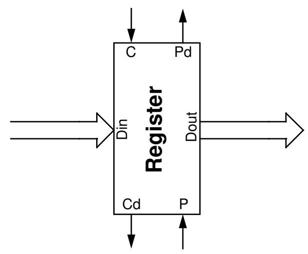


Figure 4.8: The Capture-Pass Storage Element

4.4.2 Event Controlled Storage Element

In his "Micropipelines" lecture, Sutherland also introduced the *Capture-Pass* latch, a storage element suitable for use in micropipelined systems. A high level view of the Capture-Pass element is depicted in figure 4.8.

The latch is controlled by two control signals, namely *Capture* (C) and *Pass* (P). Initially the latch is in its *transparent* state, where the input is connected through to the output (i.e. $Din = Dout$). When an event is issued on the Capture wire (C) the input-output connection is interrupted, the data is "latched", and an event is issued on the Cd signal (Capture done) to indicate the change of state in the latch (i.e from transparent to *opaque*); the latched data does not change with subsequent data input changes.

When an event arrives on the Pass wire, the input is connected back through to the output, thus making the latch transparent again; this change is indicated by an event on the Pd (Pass done) signal. The Capture-Pass may repeat, with events arriving alternately on the C and P wires respectively.

Sutherland described various implementations of the Capture-Pass latch, using inverters and switches suitably connected [Suth89], pp. 727-728.

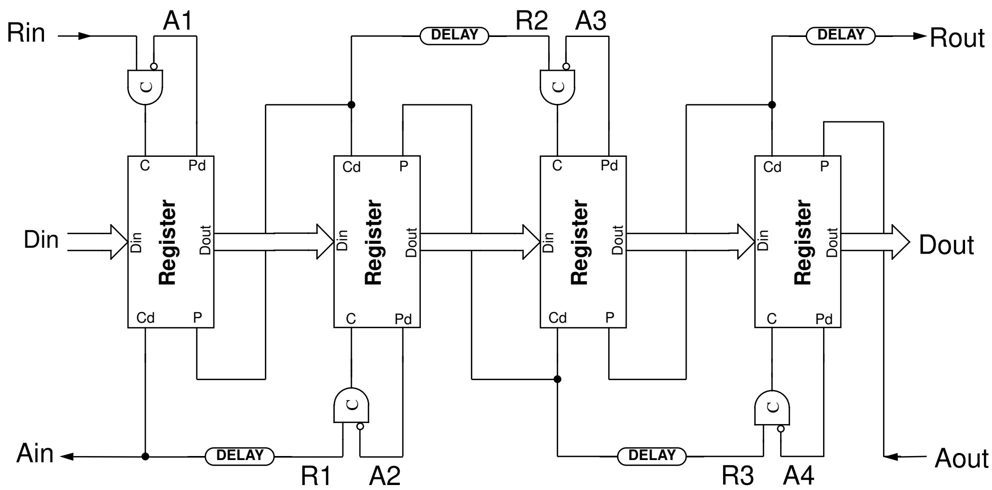


Figure 4.9: Micropipeline Without Processing

4.4.3 Micropipelines Without Processing

By combining control blocks with event controlled storage elements, Micropipelines of arbitrary complexity may be constructed. The simplest Micropipeline is a series of Capture-Pass registers connected together to form a First-In-First-Out (FIFO) structure. The registers are controlled by a series of Muller-C elements as depicted in figure 4.9. The small circles on Muller-C elements' inputs, indicate those which have been initialized in an active state, even though an event has not been received yet; thus, as soon as the first event arrives on the other input wire causes Muller-C to generate an output event.

Initially, when the pipeline is empty, the value of all the wires is Low and all the registers are transparent. When the first request event arrives on *Rin*, it passes through the Muller-C element causing the first Capture-Pass register to capture the data on the data bus (*Din*), enter its opaque state, and issue a "Capture done" event on the *Cd* wire. This event is sent as an acknowledgement on the *Ain* wire to indicate that the data has been latched and, therefore, may be removed from the data bus.

The "Capture done" event is also sent as the request signal ($R1$) to the next register of the pipeline, to indicate the availability of data to be latched by that register; this is done through a delay unit which is intended to slow down the request event and give the data enough time to arrive at the register before the request, thus guaranteeing that the bundled data delay constraint is not violated.

$R1$ will propagate through the Muller-C element and will force the second register to latch the data, whereupon Cd will be activated issuing an acknowledge event back to the first register in the pipeline. This acknowledgement will activate the Pass control wire (P) of the first register forcing it to become transparent again. The "Pass done" generated by the first register as a response is directed to the Muller-C element to enable a subsequent request event on Rin to propagate through the Muller-C and close the register.

This process continues, with the data propagating through consecutive stages until it reaches the output of the pipeline ($Dout$), whereupon a request event on $Rout$ will be issued; with the arrival of the corresponding acknowledgement on $Aout$, the data value will be removed from the pipeline. While $Aout$ is awaited, further request events arriving on $R3$ will not be allowed to propagate through the Muller-C element; thus, any subsequent data values which enter the pipeline during this time will occupy consecutive registers of the pipeline until the pipeline becomes full, whereupon no more requests will be able to enter the pipeline until $Aout$ is issued and the data value at the output is removed.

4.4.4 Micropipelines With Processing

The simple FIFO micropipeline described in the previous section can be enhanced to perform processing on the data, by interposing the necessary logic (combinatorial circuits) between adjacent register stages as depicted in figure 4.10. The operation of the pipeline is similar to that of the simple, non-processing FIFO,

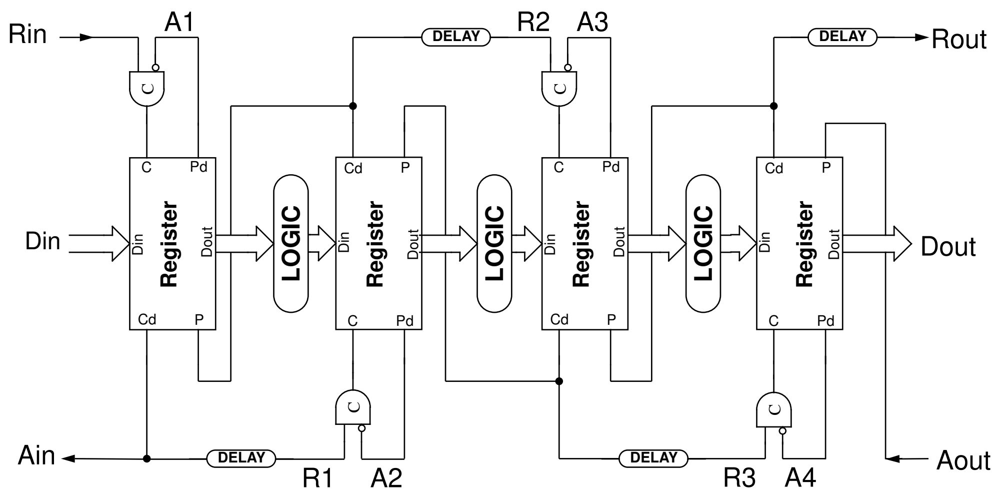


Figure 4.10: Micropipeline With Processing

but the request signals in the forward propagation path must be further delayed to accommodate the extra delays imposed on the datapath by the processing logic.

4.5 AMULET

With the aim of exploring the potential advantages of asynchronous logic and in particular, the potential for reduced power consumption, the AMULET group [AMUL] was established in 1990 by Profesor S. B. Furber in the University of Manchester. The group has been involved in a number of ESPRIT funded projects, including OMI-MAP[3] (Open Microprocessor systems Initiative - Microprocessor Architecture Project, ESPRIT project 5386), whose objective was

[3]The industrial and academic partners of AMULET within OMI-MAP included Acorn Computer Limited, Advanced RISC Machines Limited, Bull, U.K. Defence Research Agency, EO Computer Limited, IMEC in Leuven, Inmos Limited, Oxford University, Siemens AG and Thomson CSF-DOI.

to investigate trends in technology and microprocessor design in order to define standards for a European computer architecture, OMI-DE/ARM[4] (Open Microprocessor Systems Initiative - Deeply Embedded ARM project, ESPRIT project 6909), which investigated the utilization of the ARM microprocessor core in highly embedded applications, OMI-EXACT[5] (Open Microprocessor Systems Initiative - Exploitation of Asynchronous Circuit Techniques project, ESPRIT project 6143), which is investigating the application of asynchronous design techniques for electronic consumer products, and OMI-HORN[6] (Open Microprocessor Systems Initiative - Highly Optimized Reusable Nucleus project, ESPRIT project 7249), which is investigating the use of low power logic in microprocessor design; AMULET also participated in the DTI funded TAM-ARM project[7] (Transforming Architectural Models) which explored the potential use of bipolar technology for asynchronous design, and was an active member of the ACiD (Asynchronous Circuit Design) basic working group.

In order to investigate the suitability of Micropipelines for the design of complex systems, the AMULET group have developed AMULET1, an asynchronous implementation of the ARM RISC microprocessor, within the OMI-MAP project. AMULET1 has been designed to offer object code compatibility with the 32-bit ARM6 processor[8]. ARM6 is described in [Furb89] [VLSI90]; a short description of ARM6's instruction set is provided in appendix A.

---

[4]The industrial and academic partners of AMULET within OMI-DE/ARM included Advanced RISC Machines Limited, Electronica S.p.A., GEC Plessey Semiconductors, Hagenuk, Hannover University, IRIS, Manchester University and UMIST.

[5]The industrial and academic partners of AMULET within OMI-EXACT included Philips Research Laboratories in Eindhoven, Eindhoven University of Technology, IMEC in Leuven, South Bank University and EDC in Leuven.

[6]The industrial and academic partners of AMULET within OMI-HORN included ACRI, Thomson CSF, Oxford University, CTI in Patras, Greece, Inmos Limited, University of Bristol, Blue Star in UK, ACSET in Belgium and the University of Karlsruhe.

[7]The industrial partners of AMULET within TAM-ARM include Advanced RISC Machines Limited and GEC Plessey Semiconductors.

[8]Actually, a number of architectural features of ARM6 such as coprocessor instructions, the MLA (multiply with accumulate) instruction and the 26-bit mode of operation, are not supported by AMULET1.

The design and operation of AMULET1 have been described by Furber, Paver, Day, Garside and Woods in a number of publications including [Furb92] [Furb93] [Furb93a] [Furb94] [Furb94a] [Furb95] [Day92] [Day95] [Gars92] [Gars93] [Pave91] [Pave92] and [Pave92a]; a more complete and detailed description of AMULET1 is provided by Paver in [Pave94].

The next section presents a short overview of AMULET1; a further discussion of various aspects of AMULET1's operation which are relevant to this thesis is provided in chapter 6, as part of the description of AMULET1's occam model.

4.6 The AMULET1 Microprocessor

AMULET1 was designed in the period 1990-1993 and was implemented using a mixture of custom datapath and compiled control logic elements. Two silicon implementations have been developed, one fabricated on a 0.7 micron CMOS process and yielding a performance of 28kDhrystones[9], and the other on a 1.2 micron process yielding a performance of 20kDhrystones; the performance of AMULET1 is estimated as 70% of the performance of its synchronous counterpart using the same geometry and operating at 20 MHz. The power consumption of AMULET1 is similar to that of ARM6 (75mW operating at 10 MHz).

4.6.1 The AMULET1 Interface

The interface via which AMULET1 interacts with its environment is illustrated in figure 4.11, where a configuration including a memory module and an MMU (Memory Management Unit) is depicted. The interface includes an output and an input data bundle for the exchange of data with the memory, two wires to signal data aborts, two interrupt and one reset signals.

[9]A description of the Dhrystone benchmark is provided in section 8.2.

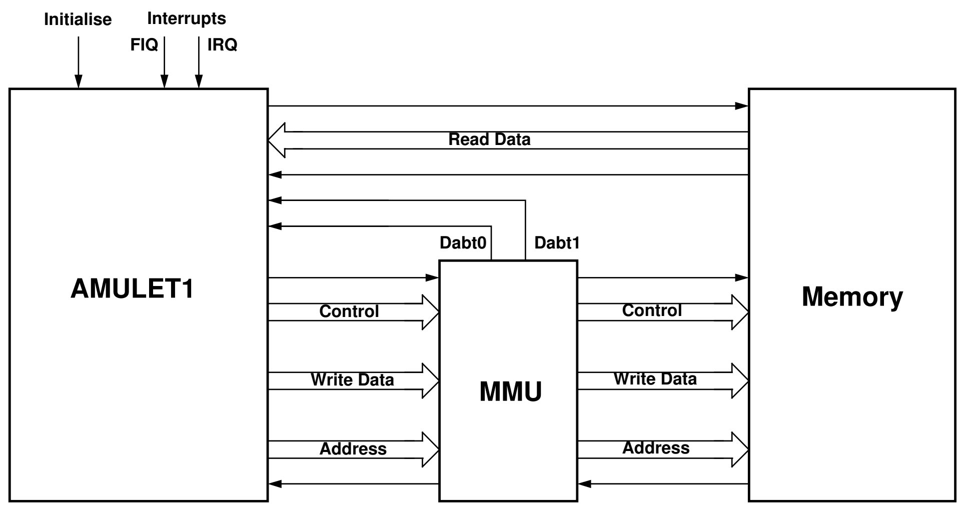


Figure 4.11: The AMULET1 Interface

The output bundle contains a memory address, control information and, in the case of a write operation, the data to be written to memory. The control information consists of a number of bits specifying the type of operation being performed (read or write), the type of information being fetched (instruction or data), the mode of operation (privilege or user) so that memory protection can be implemented and whether sequential address access is likely, so that fast page mode of DRAM may be used.

In the case of read operations, the memory responds to the processor's request by issuing a bundle with the requested information (instruction or data).

The definition of the ARM6 architecture includes support for virtual memory. Each time a data address is issued to memory, the current state of the processor is preserved until the MMU responds by issuing an abort/no abort signal indicating whether or not a page fault has occurred. If a fault occurs the exception handling software is invoked. In AMULET1, the response from the MMU is issued using a two-plus-wire protocol, (although no acknowledgement event is included in the transaction, see section 4.3.3.3); if a page fault occurs, the MMU issues an event on Dabt1 wire (abort), otherwise an event on Dabt0 (no abort) is generated.

In the case of an instruction address causing a page fault, no explicit signalling is required as the state of the processor is not directly affected; the aborted instruction is tagged as invalid and the exception handling software is invoked when the invalid instruction is detected by the processor. The operation of AMULET1 with regard to aborts is described in section 6.5.

AMULET1 also supports the two level-sensitive interrupt signals specified by the ARM6 architecture, namely *FIQ* and *IRQ*. These are completely asynchronous to the operation of the processor and may be issued at any time.

The reset signal is used to initialize the state of the processor, whereupon the issuing of sequential instruction addresses to memory, starting from zero, commences.

4.6.2 The AMULET1 Internal Organization

The internal organization of AMULET1 is depicted in figure 4.12. The processor consists of five major units, namely the address interface, the data interface, the execution unit, the register bank and the primary decode. Figure 4.13 illustrates the layout of the functional units of AMULET1 on the 1.2 micron implementation of the processor.

4.6.2.1 The Address Interface Unit

The address interface is responsible for providing all address information to memory. It operates as an autonomous unit, issuing sequential instruction addresses to maintain a steady flow of prefetched instructions to the processor.

Data transfer and branch target addresses, generated by the execution unit of the processor, are also issued to memory through the address interface unit, temporarily interrupting its autonomous operation. For multiple data transfer

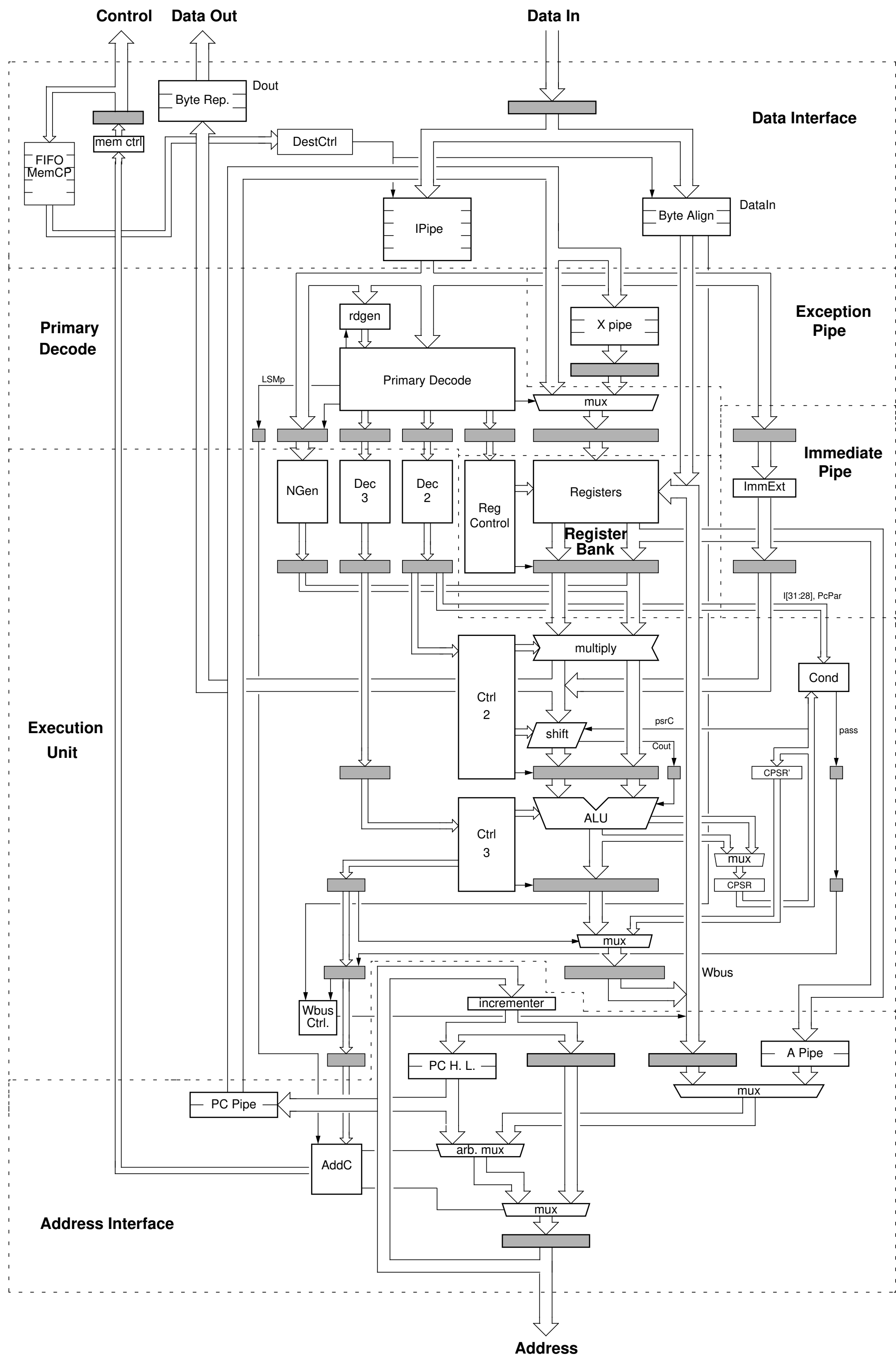


Figure 4.12: The AMULET1 Internal Organization

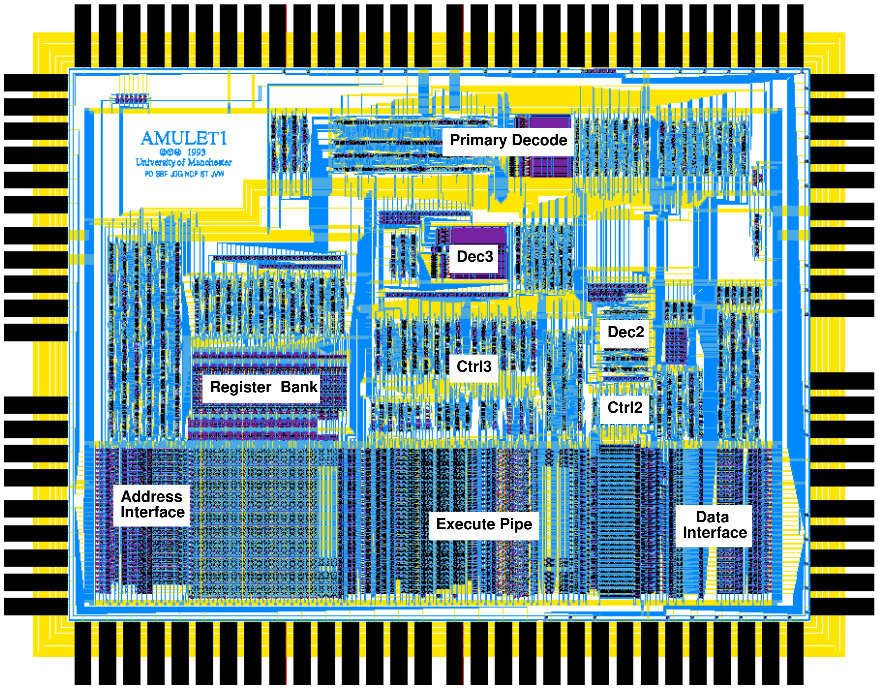


Figure 4.13: The AMULET1 Processor Physical Layout

instructions[10], the Incrementer module is activated to generate appropriate sequential memory addresses.

The address interface is also responsible for providing the execution unit with the appropriate Program Counter (PC) value (via the *PC Pipe*), to be used as the general purpose register R15 for the execution of the current instruction, as specified by the definition of the ARM architecture. For instructions which have the potential to abort, the PC value is also copied into the *Exception Pipe* (*XPipe*) to be preserved.

The address interface unit is described in more detail in section 6.3.

4.6.2.2 The Data Interface Unit

All data flow between the processor and the memory subsystem is channeled through, and controlled by, the data interface. A memory access via the data

[10] This refers to LDM and STM instructions, see appendix A.

interface may be either a *write* or a *read* operation; the latter concerning data values or instructions.

Read addresses are simply forwarded to the memory. The data interface keeps a record (in *MemCP* FIFO) of the type referenced by the address (i.e. data or instruction), so that the value returned by the memory will be directed (by *DestCtrl* module) to its correct destination; instructions are stored in a FIFO (*IPipe*) before they are executed, while data values are forwarded to the register bank via *DataIn*; this may rotate the data word by byte quantities for non word-aligned loads and mask out the top 24 bits for byte reads. For instructions which use immediate values as operands, the *Immediate Pipe* (*ImmPipe*) is activated to extract the immediate value from the instruction word; ImmPipe also performs sign extensions where required.

For write operations, the write address from the address interface needs to synchronize with the value to be written before it is forwarded to the memory; this value originates from the execution unit of the processor.

Before pairing up with their associated addresses, data values pass through *Dout*, a three stage pipeline which incorporates control logic performing an optional byte replication operation to enable byte writes to any byte-aligned address.

The operation of data interface is further discussed in section 6.4.

4.6.2.3 The Register Bank Unit

The register bank provides the top level of the memory hierarchy by means of thirty general purpose and five status registers (Saved Processor Status Registers, SPSRs). The register bank is the central structure of the processor's datapath. The execution of a typical instruction by the processor involves a) reading the contents of the source registers[11] specified by the instruction operands, b) performing the operation dictated by the opcode, and, c) sending the result back to the register bank to be written in the instruction's destination register.

The register bank includes mechanisms for dealing with multiple pending write operations and instruction interdependencies, issues which may cause problems in an asynchronous environment.

A more detailed description of the operation of the register bank is provided in section 6.7.

4.6.2.4 The Execution Unit

The execution unit is the computational core of the processor comprising a multiplier, a shifter and an ALU. It is organized as two major sub-units, referred to as *Decode2* (*Dec2-Ctrl2*) and *Decode3* (*Dec3-Ctrl3*) which control the operation of the multiplier/shifter and the ALU respectively. The multiplier involves a shift-and-add operation using carry-save adders. It operates on two source operands and yields a partial product and carry output which have to be forwarded to the ALU for the calculation of the final result. The shifter is based on the ARM6 barrel shifter and is connected to one of the operand buses, in series with the ALU. The asynchronous ALU performs all the logical and arithmetic operations specified by the ARM architecture, namely *XOR*, *AND*, *OR* and *addition*. Results produced by the ALU can be placed onto the write bus (*WBus*) to be transferred to the register bank or to memory via the address interface. The execution unit shares the write bus with the data interface which uses it to transfer data arriving from memory. Since values from the execution unit arrive asynchronously with relation to the incoming data, arbitration is necessary to enable the sharing of the write bus between the two data streams.

[11] The maximum number of source registers being accessed at any particular moment is dictated by the number of output ports in the register bank which in ARM6 is two.

The execution unit also includes logic for the control of the Current Processor Status Register (CPSR).

Section 6.8 provides a more detailed discussion on the operation of the execution unit of AMULET1.

4.6.2.5 The Primary Decode Unit

The primary decode constitutes the entry point to the datapath of AMULET1. This is where instructions arriving from memory are first decoded before they proceed to be executed. The primary decode produces the signals which control the operation of the register bank and performs an initial partial decoding for the execution unit. The primary decode unit is described in section 6.6.

4.6.3 AMULET2

The prototype AMULET1 has demonstrated the feasibility[12] of employing asynchronous logic for building large and complex asynchronous systems, but it has failed to achieve the performance and power efficiency promised by asynchronous logic.

Thus, the AMULET group have commenced the design of AMULET2, a new improved version of AMULET1, within OMI-DE/ARM project. The objective of AMULET2 design is to apply technological improvements (e.g. alternative latch designs, use of four-phase protocol etc.) and architectural enchancements (e.g. improved pipeline sizes, incorporation of a "Last Result Register", branch

[12] Although the first asynchronous processor was developed by Martin et al. at Caltech [Mart89a] [Mart89b], and various other asynchronous microprocessor designs were proposed in the late 1980's and early 1990's [Davi89] [Gino90] [Dean92], AMULET1 was the first Micropipelined processor to address critical and essential issues such as hardware interrupts and exact exceptions. More recently, following AMULET1's construction, a number of asynchronous microprocessors have been developed, including the *Counterflow Pipeline Processor*, by Sutherland, Sproull, Molnar et al. at Sun Labs [Spro94], the *NSR* RISC processor, by Brunvand et al. at the University of Utah [Brun93] and the *TITAC* processor, by Naya et al. at Tokyo Institute of Technology [Nany94].

prediction mechanisms etc.) in order to increase instruction throughput while reducing power consumption [OMI94].

4.7 Summary

This chapter has discussed issues related to asynchronous hardware systems. The various approaches to designing asynchronous systems have been presented, and a description of Sutherland's "Micropipelines" has been provided. The last part of the chapter has provided a short description of the AMULET1 asynchronous microprocessor.

The next chapter concentrates on the modelling and simulation of asynchronous hardware systems.

Chapter 5

Modelling Asynchronous Systems

5.1 Introduction

The revival of interest in asynchronous digital logic has revealed a strong need for suitable techniques for modelling asynchronous systems.

Recent years have witnessed the development of an increasing number of different methodologies for the design, automatic synthesis and verification of asynchronous circuits. Typically, these methodologies employ different notations and techniques for the specification and description of asynchronous designs. A detailed description of asynchronous design methodologies is beyond the scope of this thesis. In depth surveys of existing asynchronous methodologies may be found in [Broz89] [Gopa90] [Hauc93] [Broz95] and [Davi95] where comprehensive bibliographies are provided; additionally, the Asynchronous Online Bibliography (async-bib@win.tue.nl) provides continuous, up to date information regarding asynchronous research.

The following section provides a short overview of the major and most influential asynchronous modelling techniques.

5.2 Modelling Techniques

Most, if not all, existing asynchronous modelling techniques are intended for the automatic generation of asynchronous circuits from high level specifications.

I-Nets, is a graphical notation for describing asynchronous systems, proposed by Molnar [Moln83]. I-Nets are based on Petri nets with transitions being labeled with signal names; firing a labeled transition models a change in the corresponding signal in the circuit. By applying a series of transformations, an I-Net may lead to an Interface State Graph which may be used to obtain a Karnaugh map and finally the asynchronous circuit; algorithms for the realization of these transformations may be found in [Spro86], pp. 7.23-7.27.

Another graphical, Petri net based notation, is the Signal Transition Graphs (STGs) developed by Chu [Chu85] [Chu86] [Chu86a] [Chu87]. STGs may be transformed to asynchronous circuits using a procedure similar to that used for I-Nets; techniques for the transformation of STGs have also been developed by Lin and Lin [LinK91] [LinK92] [LinK92a], Meng et al [Meng89], Vanbekbergen et al. [Vanb90] and Yakovlev [Yako92].

State transition diagrams have also been used as a specification notation for the automatic synthesis of asynchronous finite state machines. Davis et al. [Coat93] [Davi95a] have developed a collection of tools (known as MEAT) which generate schematic diagrams at the complex gate-circuit level from state transition diagrams. A similar approach for the automatic synthesis of asynchronous finite state machines has been used by Nowick, Dill and Yun [Nowi91] [Nowi91a] [Yun92] [Yun92a].

5.2.1 CSP-based Modelling Approaches

The "Communicating Sequential Processes" (CSP) model of computation has attracted the interest of many researchers as a potential means for the modelling of asynchronous designs due to the strong relationship between its semantics and the behaviour and structure of asynchronous systems:

- CSP supports a concurrent, process-based, asynchronous, non-deterministic model of computation which exactly matches the behaviour of hardware built using asynchronous logic.

- In CSP, the communication between different modules is point-to-point, synchronous and unbuffered. This behaviour directly reflects the interaction between subsystems in asynchronous hardware, where a sender and a receiver rendezvous before physically exchanging data via wires, which are memoryless media.

Several asynchronous modelling techniques have been developed which use CSP-based notations.

A research group in Eindhoven University [Eber91] [Rem83] have developed a formalism called *trace theory* whereby, starting from high level specifications expressed in a CSP-like mathematical notation, circuits may be derived via transformations (which are referred to as *commands* by Ebergen).

A variant of trace theory has been used by Dill to verify asynchronous circuits [Dill89].

Udding and Josephs [Jose90] [Jose91] have adopted an algebraic approach whereby, a system specification expressed in a CSP-like notation may be transformed into a circuit via the use of a set of lemmas and axioms, known as *Delay-Insensitive algebra*. Employing this method, a stack, a routing chip and an up-down counter have been developed.

Martin [Mart86] [Mart89] [Mart90] and Burns [Burn87] have developed compiler-oriented techniques for the translation of specifications expressed in CSP into four-phase delay insensitive circuits. Their techniques have been used to develop a number of circuits [Mart85] [Mart85a] [Mart85b], including a complete asynchronous microprocessor [Mart89a] [Mart89b].

Akella and Gobalakrishan have extended Martin's work in a system called Shipha [Akel91] [Gopa93] which also employs a CSP-based notation but allows global shared variables.

Van Berkel's group at Philips Research labs (Eindhoven) have developed *Tangram*, a CSP-like language, for the specification of asynchronous systems [VaBe91]. A Tangram program can be compiled by syntax oriented translation [VaBe88] [VaBe88a] [Nies88] into an intermediate form, which is referred to as a *handshake circuit* [VaBe92]. A handshake circuit is a network of asynchronous components, the *handshake processes*, which communicate via channels. Handshake processes are directly mapped onto VLSI implementations.

Brunvand and Sproull [Brun89] [Brun91] [Brun91a] employ an occam-like language to describe asynchronous systems. An occam-like specification can then be compiled into an intermediate form using syntax directed translation, which after peephole optimization can be mapped onto a library of transition signalling components.

5.3 Modelling Micropipelined Systems with Occam

Contributing to the quest for suitable modelling notations and techniques for asynchronous systems, and triggered by the increasing debate regarding the potential use of CSP for this purpose, part of the research presented in this thesis investigated the suitability of the occam programming language for modelling asynchronous systems. The investigation targeted asynchronous systems that are based on Sutherland's Micropipelines, however the results may also be applied to other asynchronous design methodologies.

5.3.1 Why Occam

There are several factors which advocate the candidacy of occam for the construction of models of asynchronous systems:

- As explained in section 2.6.1, occam forms a practical realization of CSP, and, consequently, maintains the strong relationship with regard to communication and computation between CSP and asynchronous systems (see section 5.2.1).

- Occam allows explicit description of parallel as well as sequential computation. This explicit control of concurrency which extends down to the command level, along with its simple but powerful syntax and "send" and "receive" commands, makes occam ideal for describing digital systems; indeed, occam has been employed for modelling digital systems at various levels by a number of researchers including Welch [Welc87], Dowsing [Dows85], Chiu et al. [Chiu94] (gate level), de Almeida [Alme94] and Neto [Neto91].

- Occam is primarily a general purpose programming language which may be executed on a computer (transputer). Thus, a specification developed using occam is automatically an executable simulation model of the asynchronous system.

- Occam is a parallel programming language and thus may be used to perform distributed simulation[1]. A simulation model written in occam may be distributed on a transputer network and execute concurrently to achieve high

[1] Occam has indeed been used as a programming language for building both conservative and optimistic distributed simulations [Xu89] [Nevi89] [Djan89] [Cai90a] [Alon93].

performance. Asynchronous hardware systems are an excellent candidate for distributed simulation. The concurrent operation of the different subsystems of an asynchronous system, the inherent parallelism within each subsystem and the lack of any global synchronization, are characteristics which support the concurrent execution of events in a simulation model. In his *flashback simulation* approach [Suth93], Sutherland attempts to exploit these characteristics of asynchronous systems and allow "out-of-order" processing of events to increase simulation speed; however, his simulation retains its sequential nature, and is intended for execution on conventional von Neumann computers. In section 3.9.1, the increasing importance of simulation performance for the design of digital systems was discussed. In the case of asynchronous hardware, there is one more factor that makes simulation performance extremely important, namely the need to test the delay independence of designs with regard to deadlocks.

5.3.1.1 The Deadlock Problem

The concurrent nature of asynchronous hardware systems along with the absence of global synchronization, introduces a problem, common in asynchronous, parallel structures, namely *deadlocks*. Deadlock is a high-level issue of the design, and occurs when the system, as a result of a particular sequence of events, reaches a state wherein at least one sub-system becomes indefinitely blocked.

In general, the sequence of events in an asynchronous system is non-deterministic. This is due mainly to the behaviour of the arbiters. As explained in section 4.4.1, an arbiter will service request events in arrival order. If two requests arrive at the same time, the choice will be non-deterministic. Asynchronous logic allows variable delays within the different sub-systems, which will affect the order in which independent request events arrive at the arbiters of the system. The correct functionality of the asynchronous system should not depend on the ordering of independent streams of events; a correct design should be deadlock free for all possible combinations of events.

Verifying that a concurrent, asynchronous structure is deadlock free is a complex and difficult issue. Substantial research effort has been invested to develop formal methods which guarantee deadlock freedom [Sifa80] [Rosc86] [FDR93]. However, existing formal techniques are not yet mature enough to tackle systems of the complexity of asynchronous computer architectures [Furb94b], although research is ongoing in this area [Birt94a] [Nick95].

A different approach is to guarantee deadlock freedom by construction, namely, by applying certain rules during the design of the system [Welc93]. However, the applicability of this approach for the design of asynchronous systems has not yet been investigated[2].

In practice, it is generally possible to identify, and thus avoid, certain design decisions that are susceptible to deadlock [Pave94]. However the size, complexity and the non-deterministic behaviour of asynchronous hardware systems do not allow intuition to guarantee a deadlock free design.

Simulation can be an invaluable aid for this problem. The approach is to run the simulation model of the system many times, each time with a different set of delays in the component sub-systems [Furb95]. Changing the internal delays of the sub-systems, changes the order in which events are generated. Consequently, the order in which events from different data streams arrive at the arbiters also changes. Since delays dictate event orderings, following this approach, the design can be tested for possible deadlocks. The degree of confidence that a design is deadlock free is proportional to the number of runs of the simulation model. The speed of simulation here is crucial; a fast simulator, would allow the

[2]Such a task, which would attempt to exploit the characteristics of asynchronous hardware systems in order to establish design rules that guarantee deadlock freedom would be extremely important and would contribute enormously to asynchronous digital system design.

delay independence of the system with regard to deadlocks to be rigorously and extensively tested for a large number of possible combinations of events.

This technique requires a modelling approach which would allow the rapid production of executable models of the architecture at a high level so that possible deadlocks are located at an early stage of the design process.

5.3.2 The Modelling Philosophy

The main objective of the modelling philosophy proposed in this thesis is to exploit:

1. The strong relationship between CSP (and occam) and asynchronous hardware, in order to achieve easy and rapid construction of models.

2. The inherent parallelism of the hardware, in order to achieve high simulation performance.

The latter may be exploited at any level of abstraction at which the system is modelled. The former, however, may be exploited only at the Register Transfer, or higher, level of a Micropipelined system.

Assuming a correct implementation of the communication protocol, at the Register Transfer Level, a Micropipelined system may be viewed as a network of concurrent modules communicating via synchronous, unbuffered communication. The modules are data-driven; each module will start computation as soon as data is available on its input wires, and will signal when its result has been computed. At this level, the system may be directly modelled using the "Logical Process Paradigm" described in section 3.4; the model will consist of a network of concurrent, communicating occam processes, topologically identical to the asynchronous system, with each occam process modelling the behaviour of a different functional module.

Since the correct operation of an asynchronous system does not depend on a global clock, simulated time is not required for the synchronization of the occam processes of the model. Processes are entirely data-driven and self-scheduling; they are synchronised by the protocol employed in the communication semantics of occam, in the same way that the communication protocol employed in the asynchronous system synchronizes the different functional modules. Each process will always consume event messages as soon as they become available, and will always wait for subsequent messages if the messages it has generated have been successfully forwarded.

This methodology provides a natural way for modelling an asynchronous system based on the similarities between the system's behaviour and the semantics of occam. This basis, however, is not available for modelling at lower levels of abstraction. In this case, no assumptions should be made regarding the correctness of the communication protocol in the system; instead, explicit modelling of the protocol to verify that it adheres to the bundled data delay constraint is required. Occam may still be used for the description of low level circuit elements (event control elements and gates), however simulated time is essential for the synchronization and the correct operation of the simulation model.

5.3.3 Modelling a Pipeline Without Processing

Following the modelling philosophy described in the previous sections, a register in a Micropipeline without processing may be modelled as depicted in figure 5.1.

The request and acknowledge signals in the circuit are used to synchronize the register with its neighbouring registers in the pipeline. In the model, synchronization between occam processes is performed by the communication protocol specified by the occam channel. Thus, no extra channels are required for the

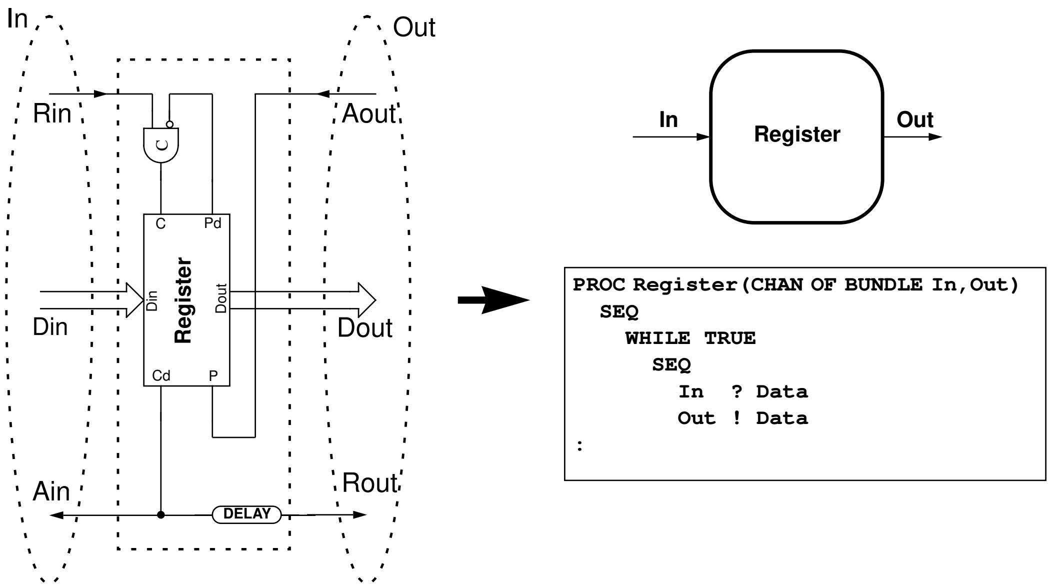


Figure 5.1: Micropipeline Without Processing: The Register Model

request and acknowledge signals. The register model makes use of two channels, for input and output respectively. The register process repeatedly reads data from its input channel and forwards it to the next process in the pipeline before it reads the next input value, thus manifesting a behaviour similar to that of the Micropipeline stage described in section 4.4.3. A multi-stage Micropipeline may be modelled by means of a parallel replication (PAR) of the register process, as described in section 2.6.1.4.

5.3.4 Modelling a Pipeline With Processing

At the Register Transfer Level, a general Micropipeline with processing may be viewed as depicted in figure 5.2. The sending register outputs its contents, consisting of data and control bits, onto the data bus and produces a request event (request wires are indicated in the figure by solid lines, while acknowledge wires are denoted by dotted lines; dotted lines appearing in figures in this thesis hereafter, denote acknowledgement paths). The control bits are used by the

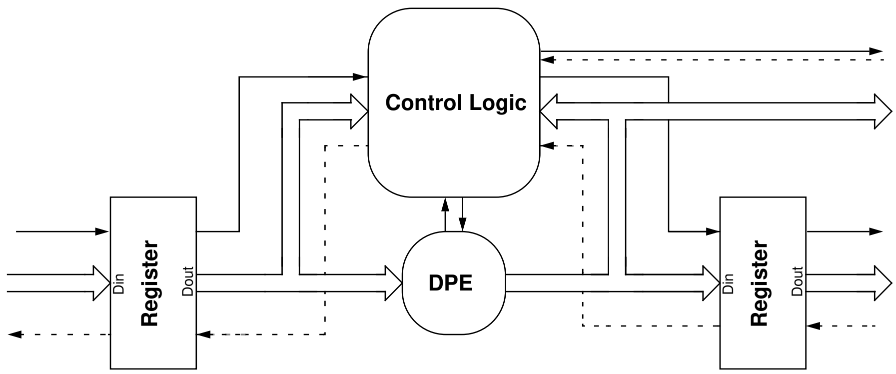


Figure 5.2: Micropipeline With Processing: A High Level View

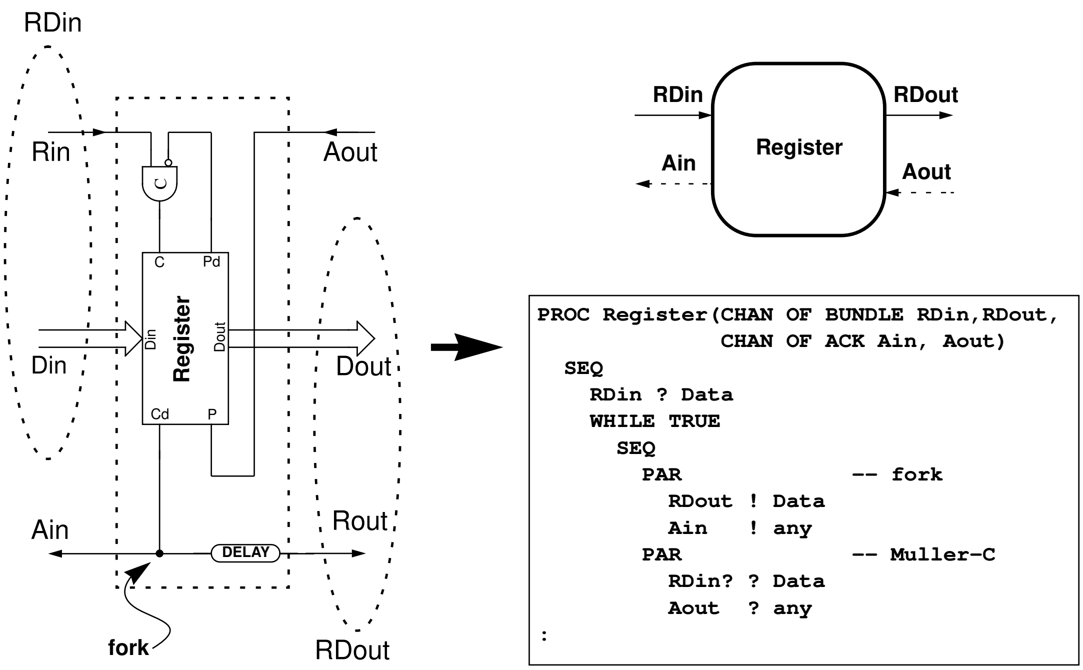


Figure 5.3: Micropipeline With Processing: The Register Model

control logic to direct the request event to its correct destination activating, if necessary, the data processing elements (DPEs, e.g. ALUs, multipliers, shifters etc.) of the circuit. Data passes through the DPEs and propagates to the next stage.

This general Micropipeline may be modelled by three occam processes, two for the registers and one for the control/data processing logic; the control logic and the DPE may be modelled as one process, with the DPE being a procedure called by the control process.

The simple register model described in the previous section, is not suitable for modelling the behaviour of a stage in a Micropipeline with processing. Using this simple register model would force the control process to act as a buffer, decoupling the register processes; the sending process would be free to read the next value from its input channel, without first ensuring that the previous value had been received by the destination register process. Thus, the control logic process would introduce an extra pipeline stage in the model, a stage that does not exist in the physical system. To avoid this situation, the register processes must be kept tightly coupled and synchronized. This may be achieved by using two channels for a communication transaction between two register processes, one for the request/data and one for the acknowledge event[3].

Figure 5.3 illustrates the generic occam register model. The model makes use of two PAR statements, one to model the Muller-C element and one to model the fork on the Ain/Rout wire. Initially the register is empty. The first value to appear on the input side will be immediately latched, activating the Cd signal and issuing an acknowledgment to the source and a request to the destination register via the Ain/Rout wire. This behaviour is modelled by reading the first input request message on RDin channel before the register process enters its main loop. Upon receiving the first input request message, the process enters its main loop where it simulates first the fork, issuing in parallel an acknowledge to the source and a request to the destination register via the channels Ain and RDout respectively, and then the Muller-C, waiting on RDin channel for a new data value to be latched and on Aout channel for the acknowledgement message indicating

[3]In his PhD thesis [Brun91], Brunvand argues that the incorporation of an extra acknowledgment channel is optional, a matter of style rather than substance; the analysis presented in this section however shows that the extra acknowledgement channel is essential if the model is to accurately describe the modelled system.

that the value previously issued to the next register process has been received by that process.

5.3.5 Modelling Control Logic

The control logic is inherently concurrent; different parts of the circuit operate concurrently while, within each part, events take place in a deterministic sequential order, i.e. the control logic implements a partial ordering of events. The simulation model should have the same degree of concurrency as the physical circuit. The control logic may be implemented as a network of communicating processes, with the occam PAR and SEQ commands being used within each process to implement the partial ordering of events of the circuit. The number of these processes depends on the degree of modularity and fidelity required in the simulation model.

Adopting a data-driven approach to model asynchronous systems, it is essential to have a mechanism for modelling the functionality and the non-deterministic behaviour of arbiters. The occam ALT construct, described in section 2.6.1.2, provides for the non-deterministic choice of messages from different channels and therefore may effectively model the behaviour of an arbiter.

5.3.6 Timing Issues

As mentioned in section 5.3.1, an occam description of an asynchronous system is automatically a simulation model which may be executed on a transputer network. The modelling methodology does not make use of the simulated time for the synchronization of the occam processes in the model. However, simulated time is still needed as a quantifier, to provide the means for the evaluation of the simulated asynchronous system. Thus, following the Logical Process Paradigm, the messages exchanged between occam processes are timestamped, while each process maintains a local clock to keep track of the simulated time. However, the distributed nature of occam and the event driven philosophy of the simulation model introduce the problem of ensuring that preemptions do not occur and the local causality constrained is not violated.

As explained in section 3.7, in distributed simulations, causality errors occur if merge processes consume and process input messages in non-increasing timestamp order. In a Micropipelined system, Micropipelines may be merged in one of the following ways:

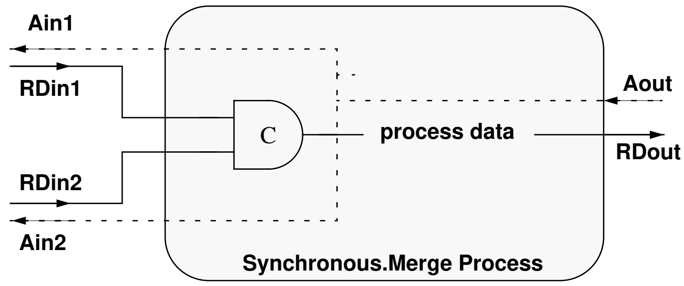
- Synchronous merge.
- Data dependent merge.
- Arbitrated merge.

5.3.6.1 Synchronous Merge

In a synchronous merge, the merge module has to wait for all input data to become available before it starts its operation. This is the case when a Muller-C element is used for the corresponding request events (figure 5.4a). In the simulation model, the corresponding occam control process has to wait for all input channels to fire. The message with the greatest timestamp is used to advance the local clock variable of the process and therefore the causality principle is preserved. The occam process which implements the synchronous merge is illustrated in figure 5.4b.

5.3.6.2 Data Dependent Merge

In a data dependent merge, the functionality of the system dictates the order in which messages from different source processes should be consumed and processed. This situation is implemented in hardware using a combination of a Select



a)

```
PROC Synchronous.Merge(CHAN OF BUNDLE RDin1,RDin2,RDout,
                       CHAN OF ACK Ain1,Ain2,Aout)
  -- data.delay : time taken to process the data
  -- ack.delay  : time taken for the acknowledge signal to propagate
  SEQ
    WHILE TRUE
      SEQ
        PAR
          RDin1 ? in1.data      --Muller-C element
          RDin2 ? in2.data
        --process data
        out.data(timestamp):= max(in1.data(timestamp),in2.data(timestamp))+ data.delay
        RDout ! out.data
        Aout ? ack
        ack(timestamp):= ack(timestamp) + ack.delay
        clock := ack(timestamp)
        PAR
          Ain1 ! ack
          Ain2 ! ack
:
```

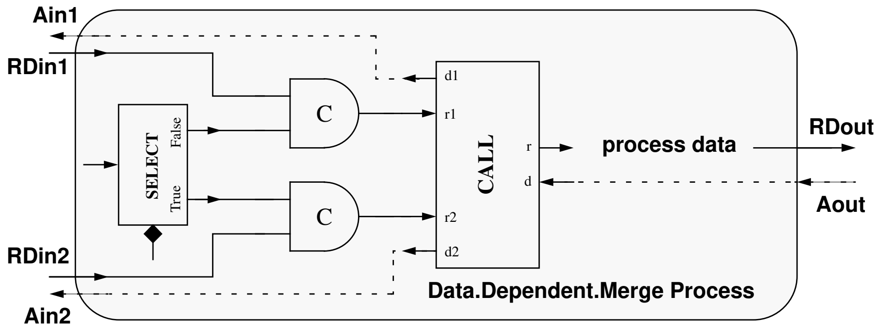
b)

Figure 5.4: Synchronous Merge

and a Call as depicted in figure 5.5a. The behaviour of the merge process in this case is similar to that of a process with just a single input channel, hence causality is not violated (figure 5.5b).

5.3.6.3 Arbitrated Merge

In an arbitrated merge, the order of request arrival defines the order of consumption. If events from two micropipelines arrive at the same time, an arbitrary choice is made. In the asynchronous circuit, arbiters are used to achieve this



a)

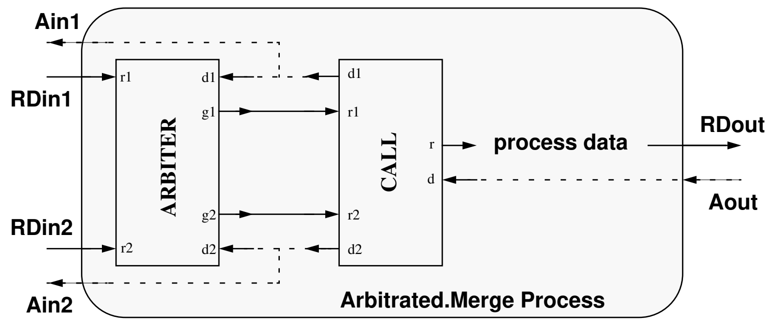
```
PROC Data.Dependent.Merge(CHAN OF BUNDLE RDin1,RDin2,RDout,
                       CHAN OF ACK Ain1,Ain2,Aout)
  -- data.delay : time taken to process the data
  -- ack.delay  : time taken for the acknowledge signal to propagate
  SEQ
    WHILE TRUE
      SEQ
        IF              --Select
          TRUE
            SEQ
              RDin2 ? in.data       -- Behaviour of a single input process
              -- process.data
              out.data(timestamp):= in.data(timestamp)+ data.delay
              RDout ! out.message
              Aout ? ack
              ack(timestamp):= ack(timestamp) + ack.delay
              clock := ack(timestamp)
              Ain2 ! ack
          FALSE
            SEQ
              RDin1 ? in.data
              --same as TRUE clause
:
```

b)

Figure 5.5: Data Dependent Merge

behaviour (figure 5.6a).

In the proposed modelling approach, an arbiter is modelled by the occam ALT construct. The order in which the ALT construct will consume messages in the simulation model does not adhere strictly to the order in which events arrive at the corresponding arbiter in the physical circuit; it depends merely on the order in which the corresponding input occam channels fire and not on the timestamps of simulated time that these messages carry. Therefore, messages may be consumed



a)

```
PROC Arbitrated.Merge(CHAN OF BUNDLE RDin1,RDin2,RDout,
                     CHAN OF ACK Ain1,Ain2,Aout)
  -- data.delay : time taken to process the data
  -- ack.delay  : time taken for the acknowledge signal to propagate
  SEQ
    clock:=0
    WHILE TRUE
      SEQ
        ALT                   --arbiter
          RDin1 ? in.data
            SEQ
              -- process data
              out.data(timestamp):= max(in.data(timestamp),clock) + data.delay
              RDout ! out.data
              Ackout ? ack
              ack(timestamp):=ack(timestamp) + ack.delay
              clock:=ack(timestamp)
              Ain1 ! ack

          RDin2 ? in.data
            SEQ
              --process in.data in a similar as above
:
```

b)

Figure 5.6: Arbitrated Merge

by the ALT construct in a non-increasing timestamp order, thus violating the local causality constraint.

This violation does not affect the correct functionality of the model; the very presence of an arbiter in the design implies that the order of consumption may be arbitrary. However, it introduces an error in the simulated time and, consequently, in the values obtained during the evaluation of the simulated system.

Despite this error introduced in simulated time, the characteristics of the simulated asynchronous systems suggest that the inaccuracy of the obtained results will be limited and indeed tolerable and acceptable at this high level of simulation. The local clocks of communicating processes will become too skewed (and thus the timing error due to a preemption large) only if the ALT construct selects the same source channel a large number of times before accepting a message from the other one (assuming that both the corresponding source processes produce request messages). The self-regulating nature of asynchronous systems, with Micropipelines acting as throttles, will however balance the throughput of the occam processes preventing, thus, the local clocks from becoming too skewed.

The results presented in chapter 8 confirm this claim; a similar approach adopted for the simulation of dataflow architectures has also produced similar results [Neto91]. Chapter 9, discusses an alternative technique for modelling arbiters, so that preemptions are avoided.

Figure 5.6b presents a description of the operation of an arbitrated merge process (an *arbiter process*). The value of the local clock variable is the timestamp of the acknowledge message forwarded by the arbiter process to the source process which issued the last request message. If t_k is the timestamp of the R_k request message currently arriving at the arbiter process on one input link, and t_{k-1} is the timestamp of the last request message that arrived at the arbiter process on the other input link, the following three cases may be distinguished:

1. $t_k >= clock$. This case indicates that the arbiter process has been idle for $t_k - clock$.

2. $t_{k-1} <= t_k < clock$. This indicates that when R_k was issued, the arbiter process was busy processing the R_{k-1} message and was thus unable immediately to consume and process it (overload situation for $t_k - t_{k-1}$).

3. $t_{k-1} > t_k$. This manifests the occurrence of a preemption as R_k should have

been processed before R_{k-1}. In this case, the current value of the clock is used by the arbiter process to calculate the timestamp of the output request message, thus ensuring that messages are issued on the output channel in increasing timestamp order (i.e. preemptions are prevented from propagating further in the simulation model).

5.3.6.4 Delay Independence

As explained in section 5.3.1.1, an asynchronous system may be tested for deadlocks by changing the order in which events are issued to arbiter processes in the simulation model. In a simulation approach where simulated time is the synchronizing force, it is the actual simulated time delays within the processes of the model which need to be modified to change the sequence in which events will occur in the simulation model.

In the proposed modelling methodology, however, the order in which an arbiter process consumes messages is completely independent of the timestamps of the messages. Hence, changing the simulated time delays of the occam processes would have no effect on the ordering of events in the model. In this case, the ordering of events may be changed by using occam "Timers" to alter the order in which processes are scheduled as discussed in section 2.6.1.3; this will change the order that processes execute and produce messages. Although "Timers" do not allow full control of the process scheduling mechanism, as the time which a process can be delayed is only approximate, this approach still allows the occam model to be used for testing the delay independence of the simulated system. By using different benchmark programs, different paths of the design may be activated. By altering the order in which occam processes are executed for a particular benchmark, the probability of an undetected deadlock condition is reduced.

5.4 Summary

This chapter has provided an overview of existing notations and techniques for modelling asynchronous hardware systems; emphasis has been placed on the techniques that employ CSP-like notations. A modelling approach has been introduced which employs occam as a description language. Finally the causality problems due to the distributed nature of the proposed modelling approach have been discussed.

The next chapter describes how the proposed modelling approach has been employed to develop an occam model of the AMULET1 microprocessor.

Chapter 6

Occarm: An Occam Model of AMULET1

6.1 Introduction

To investigate the suitability and applicability of the approach described in the previous chapter for modelling large and complex asynchronous hardware systems, *occarm*[1], an occam simulation model of the AMULET1 processor has been developed.

When the modelling work commenced, AMULET1 was in the early stages of the design process. A model of AMULET1 had already been developed using Asim, the ARM Ltd's in-house simulation language [ARM]. This is a conventional, sequential, discrete event simulation model which describes AMULET1 at a mixed gate/Register Transfer level.

The construction of occarm proved a challenging task. The complexity of AMULET1 introduced a number of design problems which forced the designers to sacrifice the purely asynchronous operation of the system in a few instances,

[1]The name of the model is derived from the combination of the words **occ**am and **ARM.**

in their effort to make the construction of the chip feasible and efficient; this presented problems to the modelling of the system using an asynchronous language such as occam.

This chapter discusses the basic functionality and structure of occarm; emphasis is given to the modelling techniques developed to address the problems imposed by the non-asynchronous operation of certain parts of the processor. A description of the internal organization of the occam processes which model the major components of AMULET1's control logic, is provided in appendix B.

6.2 Occarm General Structure

Occarm consists of more than fifteen thousand lines of occam code and describes AMULET1 at the Register Transfer Level. It executes ARM6 machine code produced by a standard ARM compiler. Instructions enter the simulator as 32-bit quantities in hexadecimal format. Instruction decoding is performed by means of PLA models implemented as two dimensional arrays of boolean values; the model makes use of a library of occam functions developed to allow instructions to be treated both as integer values and as one dimensional boolean arrays.

Occarm has been implemented as a hierarchy of occam processes, with each process modelling a different functional module of AMULET1. Its top level process structure graph is depicted in figure 6.1.

AddInt and *DatInt* processes model AMULET1's address and data interface units respectively. The datapath is modelled by four processes, namely *Decode1*, *Decode2*, *Decode3* and *RegBank*. Decode1 describes the primary decode unit while Decode2 and Decode3 model the two major components of the execution unit of the processor (see section 4.6.2.4). The RegBank process incorporates the functionality of the register bank. *WrtCtrl* models the operation of AMULET1's write bus control logic (see section 4.6.2.4).

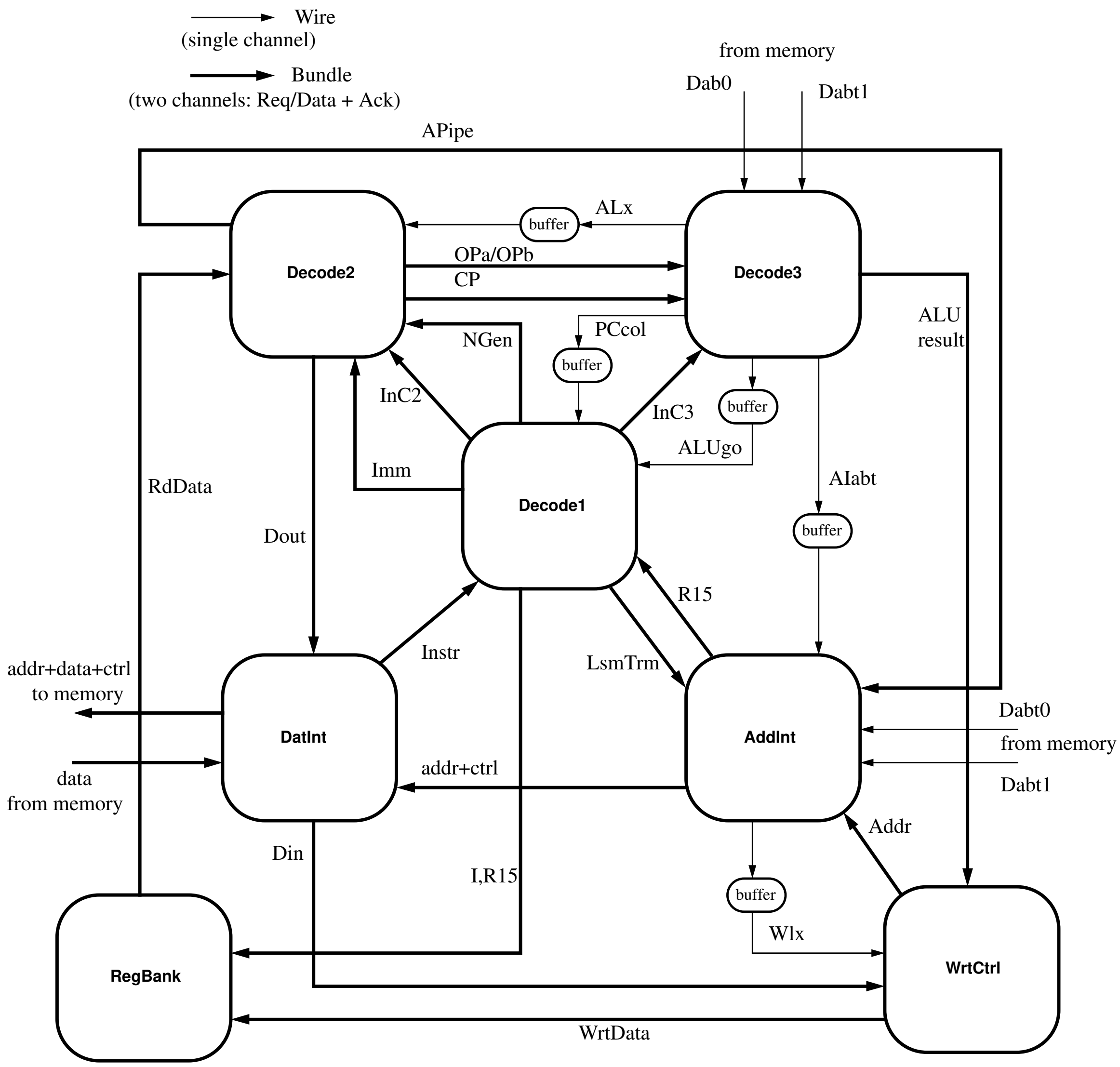


Figure 6.1: Occarm Top Level Process Graph

All the registers of AMULET1, have been modelled using the generic register model described in section 5.3.4, with interprocess communication being performed using pairs of request/data and acknowledgement channels.

```
PROC buffer()
  SEQ
    WHILE TRUE
      SEQ
        in ? data
        out! data
:
```

Figure 6.2: The Buffer Process

6.2.1 Non-Bundled Signals

To achieve code compatibility with its synchronous counterpart, it was necessary to include in AMULET1 a number of control signals which are not part of a bundled communication and do not obey the protocol specified by the Micropipelines framework. These signals are transmitted via simple wires. The modelling of these signals by simple occam channels may lead to deadlocks in the simulation model due to:

1. The communication semantics of occam, and
2. The direction of the signals.

In AMULET1 the simple wires are used to send information to previous stages of the pipeline, thus forming closed paths (loops). The synchronous communication supported by occam, forces the processes in the loop to block waiting for each other, a situation that is susceptible to deadlock.

To overcome this problem, in occarm, these signal wires have been modelled as buffer processes which hold the value of the signal at any particular moment. The buffer decouples the processes involved in the wire communication, thus eliminating the deadlock susceptible behaviour. The size of the buffer for each signal is one; this is adequate since the value on the corresponding wire will change only once for each interaction of the processes. The buffer process is shown in figure 6.2.

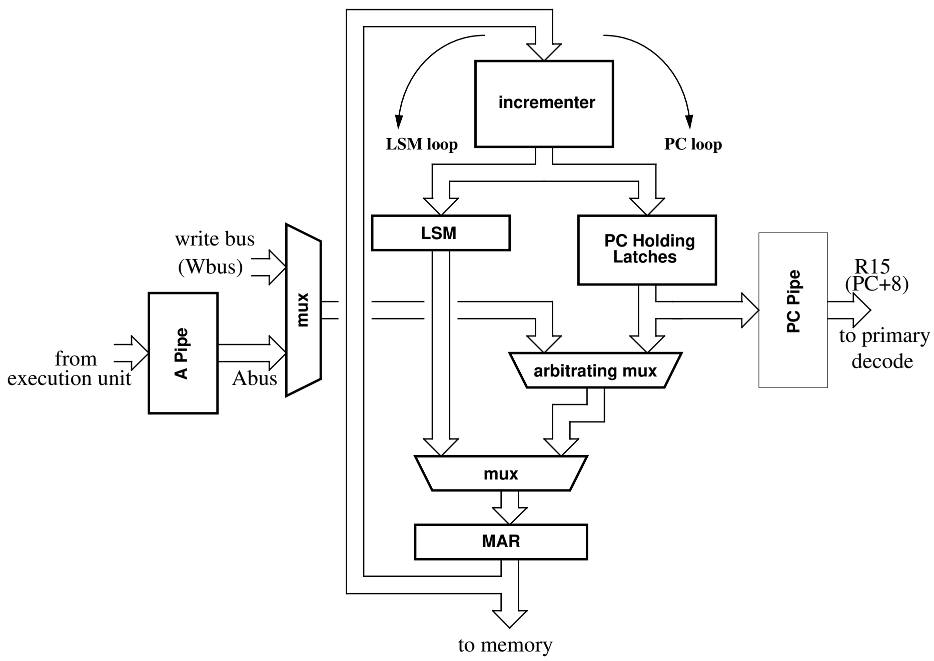


Figure 6.3: The Address Interface

6.3 The Address Interface

Before the modelling of the address interface unit is discussed, a description of its operation is provided.

6.3.1 The Address Interface Internal Organization

The internal organization of the address interface unit of AMULET1 is depicted in figure 6.3. The unit consists of three main components, namely the *prefetching loop* (also referred to as the *PC loop*), the *PC Pipe* and the *Load/Store Multiple loop* (referred to as the *LSM loop*).

6.3.1.1 The PC Loop

The PC loop is the instruction prefetching mechanism of AMULET1. It consists of the *Memory Address Register* (MAR), the *Incrementer* and two *PC Holding Latches*, connected to form a ring as illustrated in figure 6.3.

The processor begins execution upon the deactivation of the hardware reset signal, whereupon MAR's output byte address is forced to zero. This is the initial value of the Program Counter (PC) and the first address to be sent to memory. Subsequently, the PC value circulates around the prefetching loop, incrementing by four each time it passes through the Incrementer. For each cycle of the PC, the next sequential address appears at the output of the Memory Address Register and is sent out to memory, together with the associated control information as described in section 4.6.1.

This operation may be temporarily interrupted by branch target or data transfer addresses produced and sent to the address interface by the execution unit of the processor. These addresses arrive at the address interface asynchronously with respect to the operation of the prefetching loop. Therefore, arbitration is required to resolve conflicts, which might occur in the attempt of the respective event streams to gain control of the Memory Address Register.

Whenever a branch target address is produced by the execution unit, it appears on the write bus (WBus) of the processor. After an arbitration phase, it eventually gains control of the multiplexer and is forwarded to the Memory Address Register. From there, it is sent out to memory and the Incrementer to become the new PC value of the prefetching loop. The old PC value, which has been waiting in the PC Holding Latches, is discarded as invalid as it passes through the control circuitry at the input of the Memory Address Register; the mechanism used in AMULET1 to discard invalid events is described in section 6.5.

A data transfer address is supplied either by the execution unit on WBus or from the *APipe* on the *ABus*, depending on the instruction and the addressing mode it uses ([Pave94], Chapter 4). A similar procedure to that of a branch operation is followed, but here the data address is not permitted to take control of the prefetching loop; it is discarded just before entering the Incrementer, allowing the PC loop to continue its normal operation.

The presence, as well as the number, of the PC Holding Latches have been dictated by the need to resolve potential deadlock situations; these might arise as a result of a data transfer request gaining control of the arbiter immediately after a branch target address has been forwarded to the Memory Address Register, and before the old PC has been discarded. If no, or only one, holding latch was included, this sequence of events would prevent the old PC from being discarded and, consequently, the new PC from circulating around the incrementing loop.

6.3.1.2 The PC Pipe

The ARM instruction set specifies that the PC may be used as an operand. In the synchronous ARM architecture, the PC is made available to the programmer as the general purpose register *R15* (see appendix A). The value of R15 is taken directly from the prefetch unit of the ARM, and in most cases is PC+8, where PC is the current contents of the Program Counter. This value reflects the depth of the execution pipeline of the ARM6; the three stages of the pipeline (i.e. fetch, decode and execute) cause the value of R15 to be 8 bytes (i.e. two instructions) ahead of the address of the instruction being executed.

In the synchronous implementation, the depth of prefetching is deterministic and the value of R15 at the moment when the instruction reaches the ALU to be executed, is well defined. In the asynchronous environment of AMULET1 however, the amount of prefetching at any particular moment is nondeterministic. Consequently, it is not possible to define a relationship between the PC in the prefetching loop and the value of R15 associated with a particular instruction.

The technique adopted in the design of AMULET1 to overcome this problem, and thus to achieve code compatibility with ARM6, is to explicitly supply the required PC value from the prefetching loop to the execution unit. This is achieved by means of a two stage micropipeline, the PC Pipe, whose input and output are connected to the prefetching loop and the processor datapath respectively. For each cycle of the prefetching loop, a copy of the circulating value is provided to the PC Pipe, through which it is sent to the primary decode to pair up with its associated instruction arriving from memory (see section 6.5); it then accompanies the instruction through the datapath to the execution unit to be used as the R15 value, if required.

6.3.1.3 The LSM Loop

The LSM loop comprises the Incrementer, the MAR and the *LSM register*, as depicted in figure 6.3. It is activated during the execution of load/store multiple instructions. These instructions provide for the transfer of multiple data values to and from consecutive memory locations (see appendix A). During their execution only the initial data transfer address (i.e. the base address) is sent to the address interface via the write bus. Once the base address gets control of the arbiter, it is forwarded through MAR to memory. Unlike the transfer of single data however, the LSM base address is also sent to the Incrementer, to generate the next sequential transfer address. This address is sent to memory through the LSM register and the cycle is repeated.

The operation of the LSM loop is controlled by the primary decode (see section 6.6). Before initiating a cycle to produce the next sequential data address, the address interface waits for a signal from the primary decode. This signal indicates whether this is the last cycle, in which case no further incrementing takes place, and for load instructions, whether the value to be loaded is intended to be the new PC. If a new PC transfer is signalled, then the value being loaded will eventually arrive at the address interface to become the new PC value of the PC loop.

To take advantage of cache memories and fast sequential modes of DRAM, the LSM loop must not be broken during the transfer operation, to ensure that the data addresses sent out to memory refer to sequential locations. This is achieved by blocking the PC loop during the transfer and allowing it to resume its operation when the the last LSM cycle completes. Consequently, no arbitration is required for the LSM address to gain access of the Memory Address Register.

6.3.2 The Address Interface Occam Model

The asynchronous, time-independent operation of the address interface, as well as its well-defined handshake interface to its peer functional modules, make its description using occam straightforward.

The process structure of the address interface occam model (i.e. the AddInt process) is shown in figure 6.4[2]. Replicated register processes are used to implement the pipelines of the unit, while two extra occam processes, namely *AddC*[3] and *Incrementer*, are included to model the functionality of the control circuits. The interconnection pattern of the processes in the model provides for the formation of the the PC and LSM loops of the address interface.

The *MAReg* register process models the Memory Address Register of the address interface unit, and makes use of two output channels to forward each new PC value in the PC loop to memory and the Incrementer respectively. The execution of the AddInt process, and indeed of occarm, commences with the MAReg register process firing its output channels, and thus issuing address 0; this causes instruction prefetching to begin at the corresponding memory location.

The acknowledgment message to the MAReg process is sent via the Incrementer, after the synchronization of the corresponding acknowledge messages

[2]Within the context of this thesis, hereafter, in figures which illustrate occam process graphs, register models are depicted as rectangles, while processes which model control logic are shown as squares with rounded edges.

[3]See also figure 4.12.

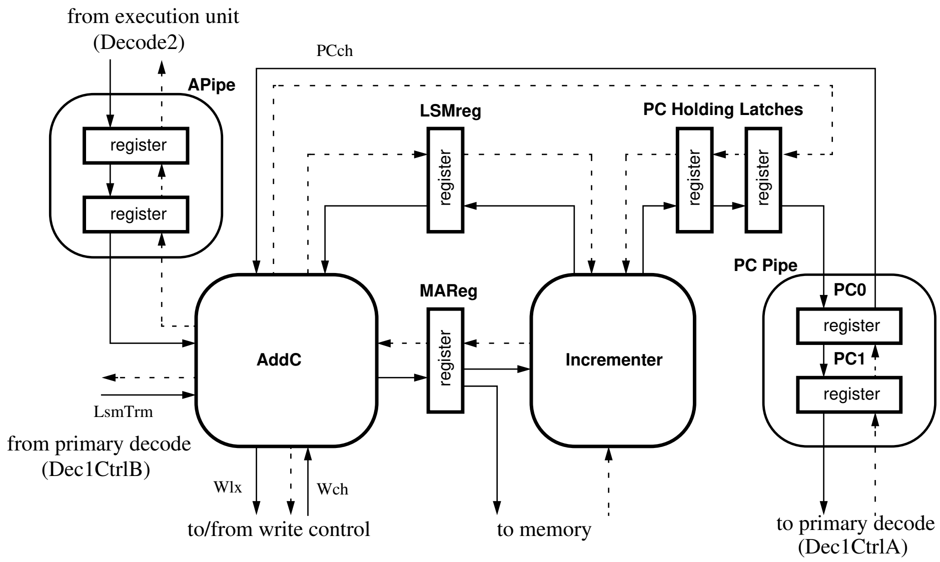


Figure 6.4: The Address interface Model (AddInt)

from the memory and the PC loop takes place. The Incrementer process also incorporates the logic which prevents data addresses from taking control of the prefetching loop (see section 6.3.1.1).

The PC Pipe has been modelled using two register processes, namely *PC0* and *PC1*. In AMULET1 the data bus at the input of the PC Pipe is also connected to the input of the MAR, thus allowing the PC value entering the PC Pipe to appear also at the input of MAR; the acknowledgement issued from the first register of the PC Pipe is directed back to the beginning of the prefetching loop to be used as a request event to the arbiter and, eventually, to the Memory Address Register. To emulate this behaviour in occarm, the acknowledgement message issued by PC0 upon receiving a new PC value, carries a copy of this value, allowing the corresponding channel to be treated as a request/data one.

The main function of the AddC process is to model the arbitration performed on the PC loop and the write bus data streams; this is achieved by employing an occam ALT construct to select arbitrarily one of the *PCch* and *Wch* channels respectively. Messages arriving on the PCch are either forwarded to MARreg or discarded (e.g. if they immediately follow a branch target address).

The *Wlx* channel is used to synchronize the address interface process with the process modelling the write bus control logic (see section 6.9); a signal is issued on Wlx channel in parallel with the acknowledgement message to the write bus control process.

The *LsmTrm* channel connects AddInt with the primary decode model (Decode1) and is used during load/store multiple operations to transmit the LSM loop terminating signal, as described in section 6.3.1.3 (see also section 6.6).

6.4 The Data Interface

Figure 6.5 depicts the process graph of the occarm DatInt process, whose structure directly reflects the organization of the data interface unit of AMULET1 described in section 4.6.2.2.

The synchronization of address messages arriving from the address interface (AddInt - MAReg) process, and data messages from the execution unit during write operations, takes place in the *MemCtrl* process. The synchronized values are then issued to memory.

The byte replication logic of the data interface, described in section 4.6.2.2, is incorporated in *Dout*, and has been modelled by a separate process, namely *DataOut*. DataOut is a single input, single output process, whose functionality may therefore be incorporated into either the source or destination processes. Such a configuration would eliminate the communication overhead introduced by the input and output channels of the DataOut process. However, in order to follow a consistent modelling strategy and maintain the generality and portability of the register processes, DataOut has been modelled as a separate process, with the acknowledgment channel bypassing it.

The same methodology has been adopted for the modelling of the DataIn

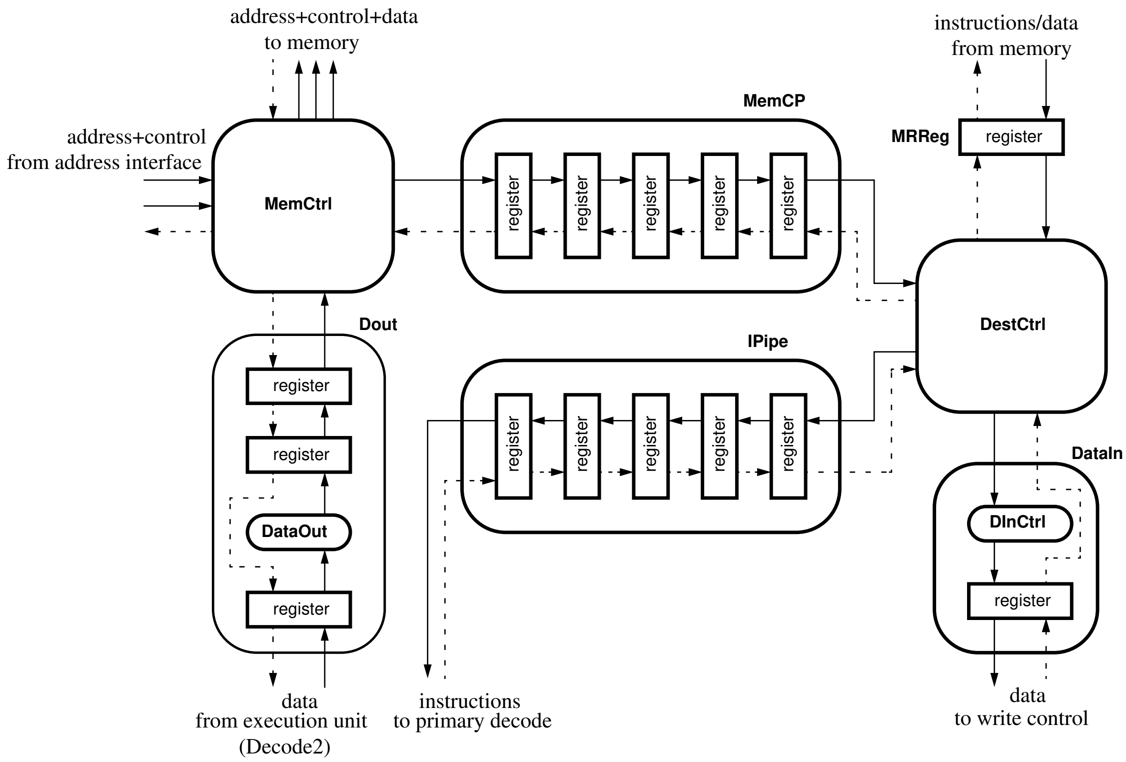


Figure 6.5: The Data Interface Model (DatInt)

module, with *DInCtrl* process incorporating the byte rotating and mask logic (see section 4.6.2.2).

For each data address sent out to memory, the MemCtrl process sends a message containing control information to the *Memory Control Pipeline* (*MemCP*). When the memory receives a read address, it responds by sending the corresponding data to the *Memory Read Register* (*MRReg*), which is then forwarded to the *Destination Control* (*DestCtrl*).

Messages arriving from the Memory Read Register, are synchronized with their corresponding control extracted from MemCP. DestCtrl uses the control information to direct incoming data to its correct destination. Instruction messages are sent through the *Instruction Pipe* (*IPipe*) to the primary decode, to be decoded and forwarded for execution. Data values are directed to the write control, either to be forwarded and stored in the register bank, or to be sent to the address interface to interrupt the prefetching loop and be used as the new PC; the latter takes place in the case of load register and data processing instructions, with R15 as destination register (see appendix A).

6.5 Instruction Flow Control

Before the modelling of the datapath of AMULET1 is described, it is pertinent to provide a short description of the flow of information in the AMULET1 processor. A more detailed discussion of the issues involved may be found in [Pave94], pp. 92-97.

In AMULET1, instructions arriving from memory through the data interface, rendezvous with their associated R15 value extracted from the PC Pipe, whereupon they enter the datapath for execution.

There are three cases wherein the execution of an instruction which has entered the processor will not take place:

- The instruction fails its condition codes in the ALU.
- The instruction follows a branch that is taken.
- An exception occurs before the instruction reaches the ALU.

The following sections examine these cases.

6.5.1 Condition Code Evaluation

To minimize the number of the branches in compiled code, all ARM instructions are conditionally executed[4]. Their execution depends on the outcome of the comparison between the *Current Processor Status Register* (*CPSR*) arithmetic flags

[4] A more detailed analysis and justification of the conditional execution of instructions in ARM is provided in [Furb89] pp. 229-230

(**N**egative, **Z**ero, **C**arry and o**V**erflow) and the condition field of the instruction word (bits [32:28], see appendix A). Typically the majority of a program's instructions will use the "always" condition, which specifies unconditional execution.

The test of the condition flags is performed by the CPSR unit of AMULET1 (see section 6.8), in parallel with the operation of the shifter and multiplier units in the first stage of the execution pipeline. The result of the comparison is used by the execution unit to invalidate and discard the corresponding instruction. For discarded instructions which have locked their destination registers, an invalid request event is issued to the register bank to unlock them (see section 6.7).

6.5.2 Branch Execution

According to the behaviour described in the previous section, only when a branch reaches the ALU will a decision be made, as to whether or not it will be taken. By that time, a number of instructions will have been prefetched and entered the processor. Since the prefetch unit is completely autonomous and decoupled from the rest of the processor, the exact number of the prefetched instructions is nondeterministic and therefore unpredictable[5].

However, prefetched instructions following a taken branch, must be discarded before instructions from the branch target are executed. Therefore, the processor must distinguish between instructions originating from the branch target, which may thus be executed, and instructions already prefetched when the branch was taken, which must therefore be thrown away. This is achieved by "colouring" the state of the processor at any particular moment. The colour of the processor, referred to as the *PCcol*, is a single bit flag which changes every time a transfer of control takes place in the processor. Each instruction address issued to memory, carries the current operating colour of the processor which will be used to mark

[5]The depth of the prefetching depends on the precise point that the interruption of the PC loop by the branch target address takes place.

the corresponding fetched instruction.

When a branch is taken, the colour of the processor changes, causing a change in the colour of instructions subsequently fetched from the branch target. The colour bit of an instruction which arrives at the datapath for execution, is compared with the current colour of the processor. If a match is found, the instruction belongs to the current valid instruction stream and is thus executed, otherwise it is discarded.

The colour of the processor changes at the ALU, after it has been decided that the branch passes its condition codes and hence it will be taken. Instructions subsequently arriving at the ALU, not matching the processor's colour, are discarded until an instruction from the new valid instruction stream (i.e. the branch target) is encountered.

To make the colour mechanism more efficient in terms of time spent and power consumed during the rejection of invalid instructions ([Pave94], pp. 95-96), each time the operating colour changes at the ALU, the new colour is also sent to the primary decode to allow colour checking at the top of the datapath and, thus, to prevent invalid instructions from entering the execution unit. The colour is transmitted to the primary decode by means of a simple wire (referred to as the *PCcol signal*) with the aid of an arbiter; the arbitration circuitry is described in section 6.6.1.

6.5.3 Exception Handling

To comply with the definition of the ARM architecture, AMULET1 supports four types of exact exceptions [Furb89] [VLSI90]:

- Software Interrupts
- Instruction Prefetch Aborts

- Hardware Interrupts, and
- Data Transfer Aborts

In AMULET1, the exception entry routine is initiated by the primary decode and is completed in three cycles. The first cycle is used to send the exception vector address to memory. The exception vector address is generated by the primary decode once it detects the exception, and is forwarded through the ALU to the address interface to become the new PC of the prefetching loop.

The specification of the ARM architecture requires that the processor should be able to restart the execution of the interrupted instruction, once the exception entry routine has been completed and the cause of the exception has been removed. To make the resumption of the execution of the interrupted program feasible, the state of the processor when the exception occurred needs to be saved; this includes the Current Processor Status Register and the return address of the instruction that was about to execute.

The preservation of the processor status is performed during the remaining two cycles of the exception entry. The second cycle copies the CPSR from the ALU into the SPSR of the corresponding exception mode (see appendix A), while the third cycle is responsible for saving the instruction return address.

Exceptions result in a change of the operating colour of the processor. The change in colour takes place in the ALU, when it detects the occurrence of the exception. Once the colour changes, a similar behaviour to the execution of branch instructions is exhibited, with prefetched instructions subsequently arriving at the ALU being discarded while the new colour is propagated back to the primary decode to prevent instructions from entering the datapath.

6.5.3.1 Software Interrupts

Software interrupts occur when executing the *SWI* (**S**oft**W**are **I**nterrupt) or an undefined instruction (see appendix A). The interrupt is detected at the primary decode, by examining the bit pattern of the instruction's opcode. The execution of the exception entry routine commences immediately after the detection of the interrupt with the generation of the exception vector address.

The required return address is obtained from the R15 value of the instruction extracted from the top of the PC Pipe.

6.5.3.2 Instruction Prefetch Aborts

Instructions which have caused a prefetch abort are marked as such by means of the *prefetch abort flag* (*Pabt*). This is a single control bit, generated by the memory system and copied into the instruction pipeline together with the aborted instruction. When the instruction enters the primary decode, the *Pabt* flag is detected, causing the instruction data to be ignored and the exception entry routine to be entered.

As in the case of software interrupts, the return address is a modified version of the R15 value of the aborted instruction arriving from the PC Pipe.

6.5.3.3 Hardware Interrupts

Hardware interrupts enter the processor via an arbiter in the primary decode and are detected in the primary decode by examining the level of the interrupt signals FIQ and IRQ, just before each new instruction, arriving from the Instruction Pipe, is allowed to enter the datapath. If an interrupt is detected, the instruction at the top of the IPipe, waiting to enter the datapath, is usurped and its execution is replaced by the exception entry routine.

As in the case of software interrupts and instruction prefetch aborts, the return address is provided by the R15 value of the usurped instruction, extracted from the PC Pipe. If the instruction immediately preceding the detection of interrupts is a branch which is taken however, then the execution of the exception entry routine is postponed until the first instruction from the branch target arrives at the primary decode.

The postponement of an exception entry following a branch is necessary to ensure that the R15 value obtained from the PC Pipe is the correct return address (i.e the branch target address); it is achieved by forcing the exception entry to adopt the colour of the usurped instruction. This causes the exception entry to be rejected repeatedly at the ALU until an instruction from the branch target with the correct colour arrives. To ensure correct operation, the interrupt signals remain active until the exception entry becomes valid.

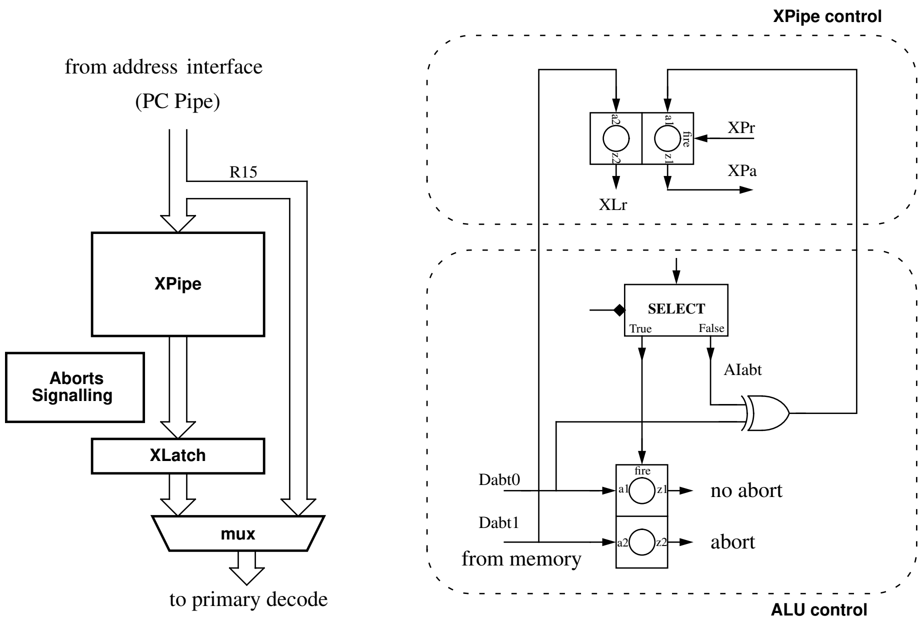
Hardware interrupts have not been implemented in the occarm model and therefore are not discussed further.

6.5.3.4 Data Transfer Aborts

The occurrence of data aborts is initially detected in the ALU rather than in the primary decode. When a data transfer instruction is executed, the ALU issues the data transfer address to memory and then it blocks until a response arrives from the Memory Management Unit (MMU) to indicate whether the transfer aborted; the MMU's response is sent by means of the Dabt0 and Dabt1 signals described in section 4.6.1.

If a data abort is signalled, the ALU changes the operating colour of the processor and sends it back to the primary decode; the detection of the new colour enables the primary decode to realize that a data abort has taken place and hence to initiate the exception entry routine (section 6.6.1).

Unlike the three aforementioned exceptions, in the case of data aborts, the



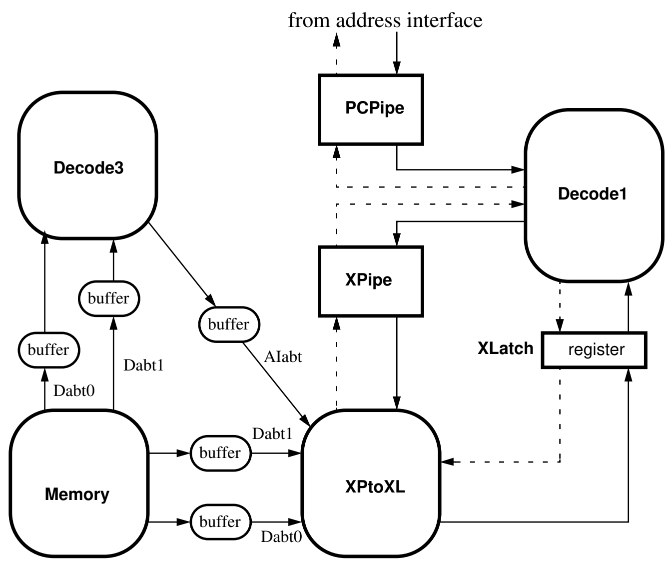
a) The Exception Pipe (XPipe) b) Aborts Signalling

Figure 6.6: The Exception Pipe

return address can not be taken directly from the PC Pipe. The PC that happens to be at the top of the PC Pipe, when the data abort is detected by the primary decode, is not the R15 value of the aborted instruction but rather, it is associated with the prefetched instruction arriving from the Instruction Pipe at that particular moment. Since the amount of prefetching is nondeterministic, the R15 value of the aborted instruction can not be computed from the PC extracted from the PC Pipe.

To preserve the return address of instructions which may cause a data abort, an extra pipeline, referred to as *Exception Pipe* (*XPipe*), has been incorporated into the AMULET1; here, the R15 values of instructions which have the potential to generate a data abort are stored.

The interaction between the XPipe and the rest of the processor is depicted in figure 6.6a. Each time the R15 value of a data transfer instruction is extracted from the top of the PC Pipe, to be sent to the primary decode, it is also copied




```
PROC XPtoXL()
  SEQ
    ALT
      Dabt0?signal
        SEQ
          --no abort
      Dabt1?signal
        SEQ
          --abort!!
      AIabt?signal
        SEQ
          --no abort
:
```

Figure 6.7: Aborts Modelling

into the XPipe. The end of the XPipe (entry point) is controlled by the primary decode; the PC value is placed on the data bus, but it is latched by the XPipe only after the decoding of the corresponding instruction indicates that it is a data transfer instruction, whereupon the primary decode issues a latch request to the XPipe.

If a data abort occurs, the value at the top of the XPipe which will be the address of the aborted instruction, is sent via the *XLatch* to the primary decode to be used as the return address; if the data transfer is successful, the value is simply discarded before it enters the XLatch. The exception PC is also discarded at the initiative of the ALU, if the corresponding instruction will not be executed as a result of it following a branch or failing its condition codes. The interaction between the XPipe and the XLatch is controlled by the abort signalling circuitry which is depicted in figure 6.6b; in the figure, *AIabt* represents the event issued by the ALU to signal the discarding of the exception PC.

Figure 6.7 illustrates the interaction between the processes in the occarm model, with regard to data aborts. The PC from the PC Pipe is sent to the primary decode (Decode1) and if necessary is forwarded to the XPipe. Two separate channels are used to read the PC either from the PC Pipe or the XLatch. Similarly, two pairs of channels are used for the abort signals produced by the memory, while an extra channel models the AIabt signal issued by the ALU. The *XPtoXL* process controls the input to the XLatch either discarding or forwarding the exception PC; it is implemented using an ALT construct which is completely deterministic since only one of the abort channels will fire for any particular instruction.

6.6 The Primary Decode

Conceptually, the primary decode unit of AMULET1 may be divided into two different sections, each one performing a distinct function. In occarm, these sections are modelled as two separate processes, *Dec1CtrlA* and *Dec1CtrlB*; these form parallel subprocesses of the Decode1 process and are connected together as depicted in figure 6.8.

Decode1 also incorporates the functionality of the Immediate Pipe of AMULET1, described in section 4.6.2.2.

6.6.1 The Dec1CtrlA Process

The modelling of the Dec1CtrlA logic demonstrated the power and usefulness of occam as an asynchronous hardware description language, as it identified parts of AMULET1, whose correct operation is not guaranteed by well defined asynchronous interfaces but depend heavily on the timing characteristics and relevant delays of the particular functional modules involved.

The basic circuitry modelled by the Dec1CtrlA process is depicted in figure

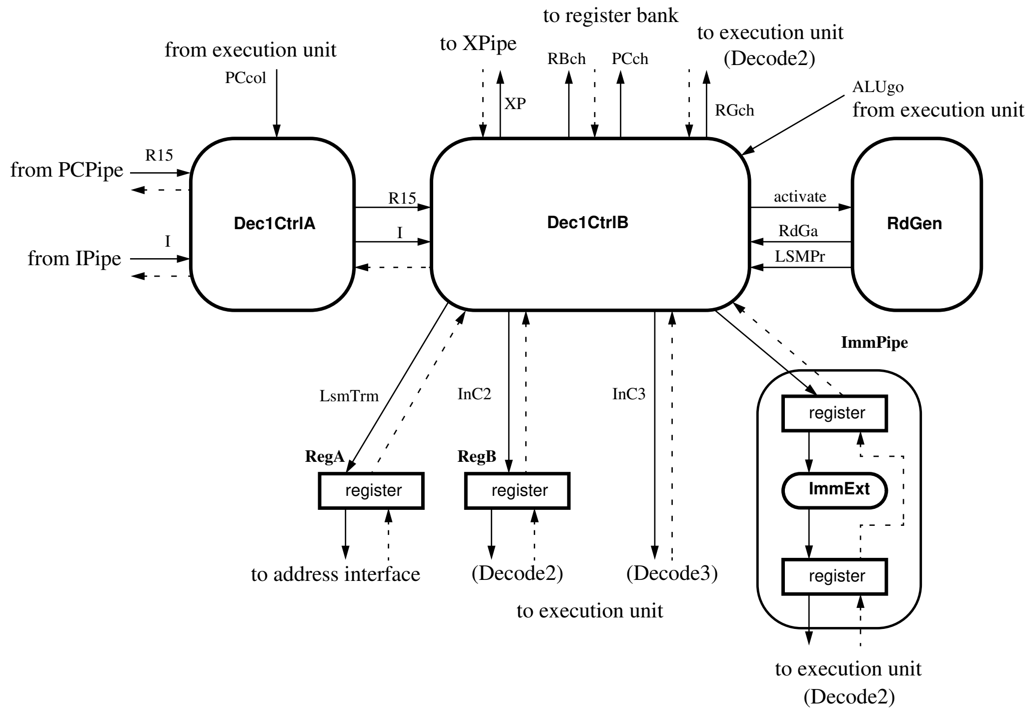


Figure 6.8: The Primary Decode Model (Decode1)

6.9. This, incorporates the logic for discarding instructions based on the colour information received by the ALU; it is also responsible for issuing the exception vector address and extracting the return address if an exception occurs.

6.6.1.1 Modelling of the Arbitration logic

In figure 6.9, part (a) of the circuit includes the arbitration hardware for the detection of the PCcol signal sent by the ALU[6]. The detection takes place just before a new instruction from the IPipe is let through to be decoded and executed.

Instructions from the IPipe (Ir) are synchronized with their associated R15 value (PCr) at the Muller-C element; the $Lreq$ event indicates that the arbiter has been locked by the instruction stream (the first $Lreq$ is generated by the reset signal Cdn). The value of $SPCcol$ indicates the current operating colour

[6]In AMULET1, this logic involves two extra arbiters for the hardware interrupt signals FIQ and IRQ, but since hardware interrupts are not supported in occarm, these arbiters have not been included in the figure.

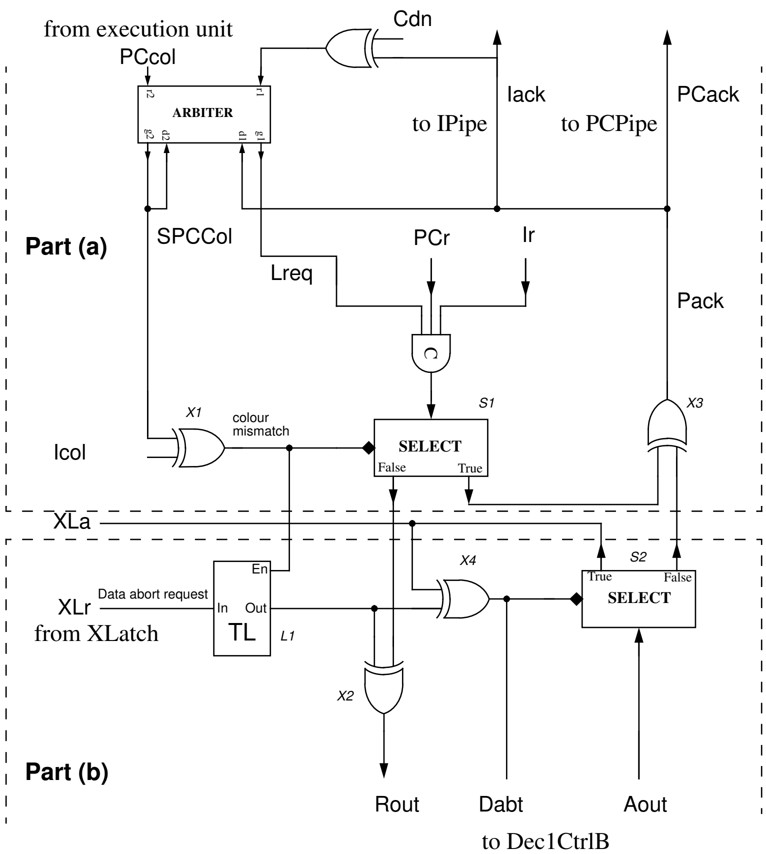


Figure 6.9: Dec1CtrlA Logic

of the processor as far as the primary decode is concerned. Whenever a new instruction arrives, its colour *Icol* is compared with the SPCcol (*Xor X1*); the bundle constraints guarantee that *Icol* will have a stable value and the comparison will have completed before the Muller-C generates its output event. If the two colours are the same, the instruction is let through to Dec1CtrlB to be decoded (*Select S1*); once the decoding is complete and the instruction has been forwarded to subsequent stages of the datapath for further processing, the acknowledgement event *Pack* is generated and sent to IPipe and PC Pipe (*Iack* and *PCack* respectively). If a mismatch is found, the "True" path is followed and the Pack is immediately generated, thus discarding the incoming instruction. The Pack signal is also used to unlock the arbiter. At this stage, if a new PCcol has been issued by the ALU and is pending, it is allowed to take control of the arbiter, hence changing the SPCcol bit; the arbiter is immediately unlocked to let more instructions enter the datapath.

An initial attempt to model the arbiter using an *ALT* statement revealed the situation illustrated in figure 6.10. A stream of instructions whose colour matches the current operating colour of the processor ("1"), are forwarded through the primary decode to be executed. The stream includes a branch instruction which, passing its condition codes in the ALU, changes the operating colour of the processor to "0". The branch target address is sent to the address interface to initiate prefetching of the new instruction stream while, in parallel, the new colour is sent back to the primary decode. If the change of the value of the PCcol wire is detected in the primary decode, while incoming instructions still belong to the old stream (figure 6.10a), correct operation is ensured; the remaining "old" instructions will be discarded as invalid while the instructions of the branch target will be let through to be executed. However, if the detection of the new PCcol takes place after the first instructions from the branch target have entered the datapath, then a number of valid instructions will be discarded (figure 6.10b).

The timing characteristics of the current design of AMULET1 ensure the correct operation of the system because the propagation delay on the PCcol wire, is less than the latency of the "ALU-memory-primary decode" pipeline. Thus, the primary decode is notified of the new colour before any instruction from the branch target arrives from the IPipe. However, any attempt to experiment with alternative designs of the AMULET1 (e.g. decreasing the number of stages in the pipeline) might alter the balance of the respective propagation delays, thus affecting the correctness of the system.

Clearly, the asynchronous, nondeterministic nature of occam cannot guarantee that the PCcol channel will fire *before* the colour of instructions on the Ir channel

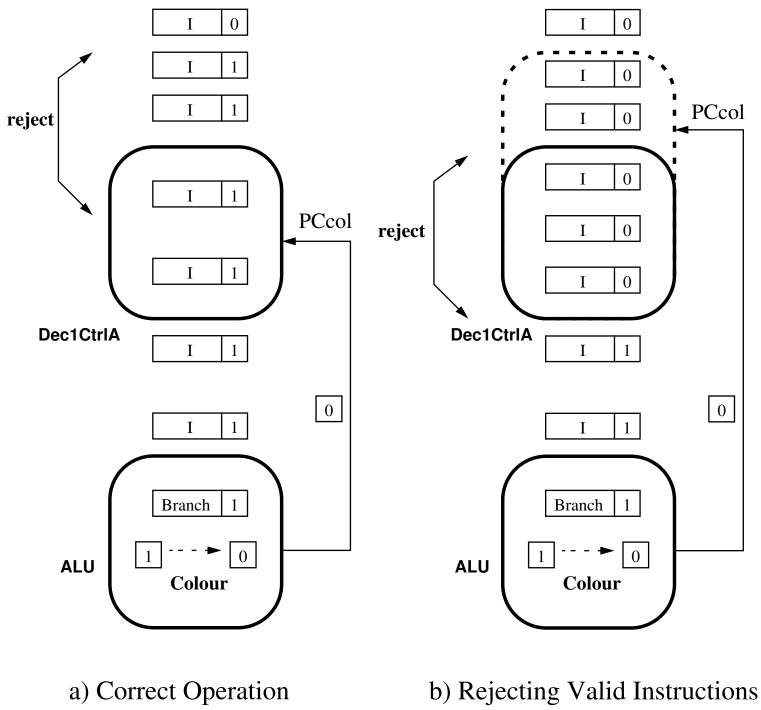


Figure 6.10: Detecting the PCcol

changes. An alternative modelling of the arbitration hardware which guarantees correct operation is depicted in figure 6.11. This model eliminates the time dependent behaviour of the PCcol mechanism by keeping a record of the colour of the last instruction which entered the system (*Previous.Icol*). If the instruction stream changes before the PCcol channel has fired, then the process blocks and waits for the new PCcol; an ALT is still used to sample the PCcol channel so that the performance of the system remains high.

6.6.1.2 Detecting Data Aborts

The rest of the logic of Dec1CtrlA (part (*b*) in figure 6.9) deals with the detection of data aborts. The primary decode needs to distinguish between the different types of exceptions, ino order to issue the appropriate vector address. Hardware and software interrupts as well as instruction prefetch aborts are detected directly by the control logic of primary decode. This is not the case with data aborts

```
PROC Dec1CtrlA()

  SEQ
    PAR
      PAR
        Ir? Instruction            -- Read Instruction and Control(Icol)
        PCr? PC                    -- Read PC from PC Pipe
      PRI ALT                      -- Sample PCcol channel
        PCcol? SPCcol
          SEQ
            SKIP
        TRUE & SKIP
          SKIP
    IF
      Icol<>Previous.Icol          -- New instruction stream
        SEQ
          IF
           SPCcol=Previous.Icol   -- ERROR!! New PCcol should have arrived
              SEQ                  -- Read it NOW!!!
                PCcol? SPCcol
            TRUE
              SKIP
          Previous.Icol:=Icol
      TRUE
        SKIP
  ...
:
```

Figure 6.11: Modelling Dec1CtrlA Arbitration Logic

however; these are initially detected by the ALU which changes the operating colour of the processor and consequently, the value of the PCcol signal (section 6.5.3.4).

A change in the value of the PCcol, however, is not adequate to inform the primary decode that a data abort has taken place, for the execution of a branch involves such a change too. The additional information required for the detection of data aborts is provided to the primary decode by the request event *XLr* of the XLatch (see also figures 6.7 and 6.6). The occurrence of an abort will result in the exception PC at the top of the XPipe being copied into the XLatch which, in turn, will generate a request event at its output. When, as a result of a colour change, PCcol takes control of the arbiter, the *L1* latch is enabled to latch the Xlr event. The latched value is used both as an event (*Xor X2*) and as a boolean (*Xor X4*) to indicate that a data abort has occurred.

Here again, the timing characteristics of the system ensure that the Xlr event will arrive while the latch is enabled. Occam can not guarantee this behaviour, for the exact time that the Xlr channel fires is nondeterministic and therefore unpredictable. A simple ALT involving Xlr (to sample its value) may yield the wrong result if an abort has occurred but the channel has not fired yet; a polling mechanism whereby the ALT is included in a loop, might result in a livelock if the colour change was not due to an abort. To make the construction of the occarm model feasible, an alternative means is required to provide Dec1CtrlA process with the necessary information and enable a decision on whether it should perform a read operation on the Xlr channel. This information may become available from the ALU, since the reason why the colour of the processor changed is known at the ALU when the new PCcol is issued. In occarm, the reason for the change is sent by the ALU as an extra bit accompanying the colour information over the PCcol channel ("0" and "1" for branches and data aborts respectively).

6.6.2 The Dec1CtrlB Process

Instructions whose colour matches the value of PCcol, are forwarded, with their associated R15 values, to the Dec1CtrlB process to be decoded (see figure 6.8). The decoding is performed by means of a PLA model invoked by Dec1CtrlB, whereupon the appropriate control messages are produced and sent to their corresponding destination processes.

The instruction's R15 value, along with the control information, is sent to the register bank via *PCch* and *RBch* channels respectively. Control messages for the execution pipeline stages are sent via *InC2* (through *RegB* register) and *InC3* channels respectively.

The *ALUgo* signal synchronizes the primary decode with the execution unit during the execution of certain multicycle instructions. The signal, which is issued in the first cycle, informs the primary decode whether the instruction will be executed and hence whether primary decode should proceed to lock the registers in the register bank; it also carries the current mode of the processor to inform the primary decode of the current accessible register set.

If the decoding points to a data transfer instruction, the R15 value of the instruction is sent to the XPipe via the *XP* channel.

For multicycle instructions, an acknowledge will not be issued to Dec1CtrlA until all cycles of the instruction have been completed. For load/store multiple operations, Dec1CtrlB initiates an interaction with the *RdGen* process; this in turn responds by issuing a message either on *LSMPr* channel to indicate that the reading/writing of registers is not complete, or on *RdGa*, to indicate completion. For each response received by the RdGen process, a message is sent to the address interface over the LsmTrm channel to indicate the destination of the next output of the Incrementer (i.e. either LSM or PC loop). Control information provided by RdGen is forwarded to the execution unit via *RGch* channel.

6.7 The Register Bank

The concurrent, asynchronous nature of AMULET1 introduces a number of hazards to the operation of the processor's register bank:

- The pipelined execution unit of AMULET1 may result in multiple pending write operations; the system must keep a record of outstanding write addresses and ensure that results are written to the correct registers.

- Instructions must be prevented from attempting to read registers pending alteration. Read operations must be blocked until all the preceding writes have completed and the corresponding registers contain valid data. The synchronization of read and write operations is a complicated issue, for the

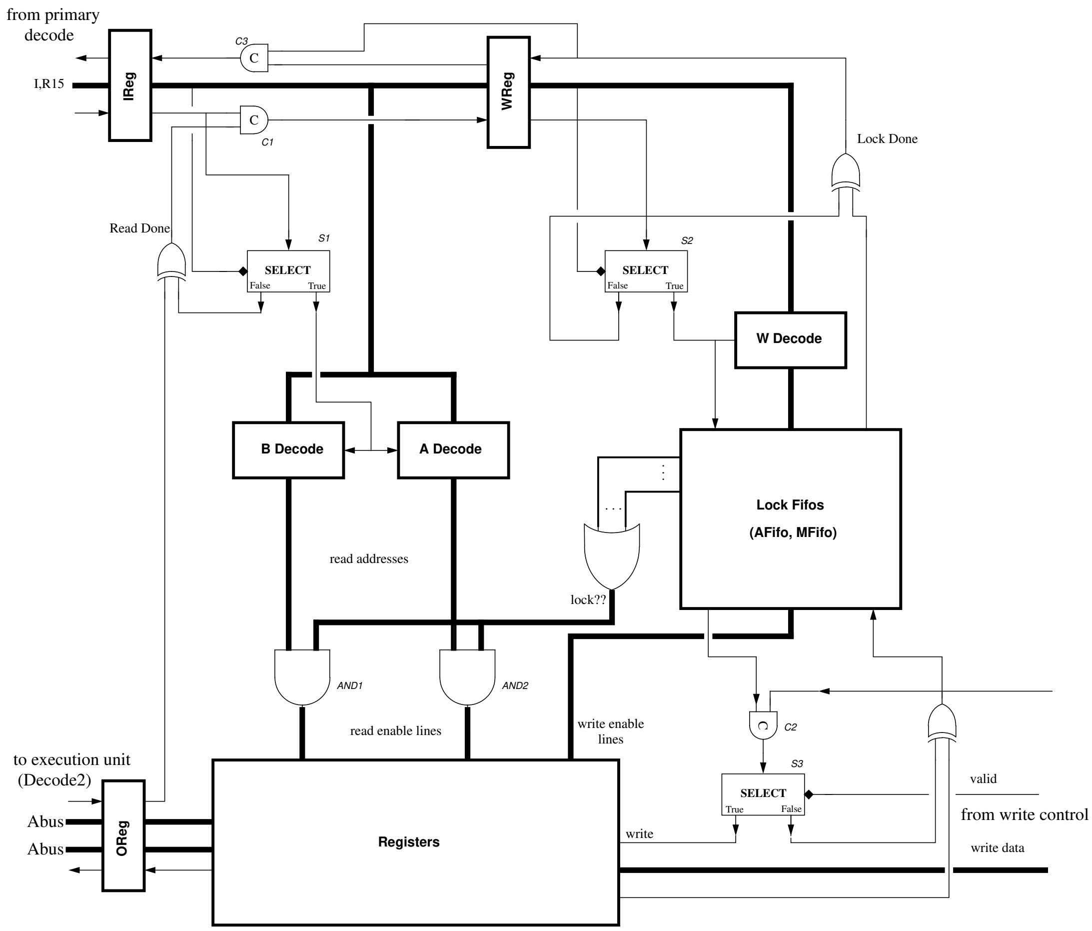


Figure 6.12: The Register Bank Internal Organization

asynchronous operation of the processor makes any assumption about their interaction infeasible.

In AMULET1, the aforementioned issues have been resolved by a novel register locking mechanism, namely the *lock FIFO* [Pave91] [Pave92]. This is an asynchronous micropipeline which holds the destination register addresses of pending write operations; these addresses are considered to be "locked", and therefore a read operation on any of them stalls until the address is removed from the lock FIFO as a result of the corresponding write operations being completed[7]. Addresses are stored in a fully decoded form as unary values, so that each stage

[7]The size of the lock FIFO dictates the maximum number of write operations that may be outstanding at any particular moment.

of the FIFO contains at most a single "1", whose position indicates the address of the locked register. The status of a register is determined simply by examining whether there is a bit set in the corresponding column of the lock FIFO; this is achieved by OR-ing together the bits of the column.

The organization of the register bank is depicted in figure 6.12. In order to allow internal results from the ALU to overtake data from the much slower memory, two lock FIFOs are used, *AFifo* and *MFifo* respectively. Decoded instructions from the primary decode enter the register bank through the *IReg* register. If the instruction specifies read registers (*Select S1*), their addresses are extracted by the *A* and *B read decoders* and are directed to the lock gating (*AND1 and AND2*); when the read enable lines are activated as a result of the corresponding registers being unlocked, the register contents are latched in the *OReg* register and forwarded to the execution unit. Once the read operation is complete, the write path is activated (*Muller-C C1*). If a write operation is required (*Select S2*), the unary representation of the destination address (*W decode*) is sent to the appropriate lock FIFO to pair with the data to be written to the register (*Muller-C C2*). If the data is invalid, it is discarded while the register is unlocked (*Select S3*). The completion of both, read and locking operations frees IReg to enable further instructions to enter the register bank (*Muller-C C3*).

6.7.1 Modelling the Register Bank

Any attempt to build a simulation model of the register bank should have the following objectives:

- To model correctly the interdependencies between the asynchronous read and write operations. This involves the accurate modelling of the behavior of the register locking mechanism.

- To exploit the parallelism within the register bank so that high performance is achieved.

Satisfying the above requirements is not straightforward. The central structure of the register bank is the lock FIFO, whose operation is based on the examination of a global state provided by the contents of its stages. The application of the logical OR on the columns of the lock FIFO creates a global view of the FIFO which makes the locking mechanism feasible and efficient.

However, within the context of the occam language, the concept of global variables between parallel processes does not exist and thus the global state of an occam system is not directly available. Therefore a mechanism is required to allow the construction of the required global view and thus, make the development of the model feasible.

A possible technique to achieve this, which has been adopted for the development of occarm, is illustrated in figure 6.13, where the complete model of the register bank is depicted. This technique requires the use of an extra process, namely *ReadLock*, whose role is to maintain a copy of the contents of the lock FIFOs at any particular moment and, based on this information, to implement the register locking.

Each time a destination address enters one of the lock FIFOs, a copy is also sent to ReadLock by the first register process of the FIFO. To ensure correct operation, the register process will not issue an acknowledgement message until the write address has been received by ReadLock. ReadLock incorporates two circular buffers, one for each of the lock FIFOs, where destination addresses are stored. Once the write operation is complete, the last register process of the FIFO involved issues a message to ReadLock to signal the removal of the write register address from the lock FIFO, whereupon the relevant entry is removed from the corresponding circular buffer. This technique allows ReadLock to include

```
PROC ReadLock()
  SEQ
    ALT
      lock?address
        SEQ
          --insert into buffer
      unlock?signal
        SEQ
          --remove top of buffer
          IF
            read address waiting
              SEQ
                --send address to READ
            TRUE
              SKIP
      read?address
        IF
          address in buffer
            SKIP
          TRUE
            --send address to READ
:
```

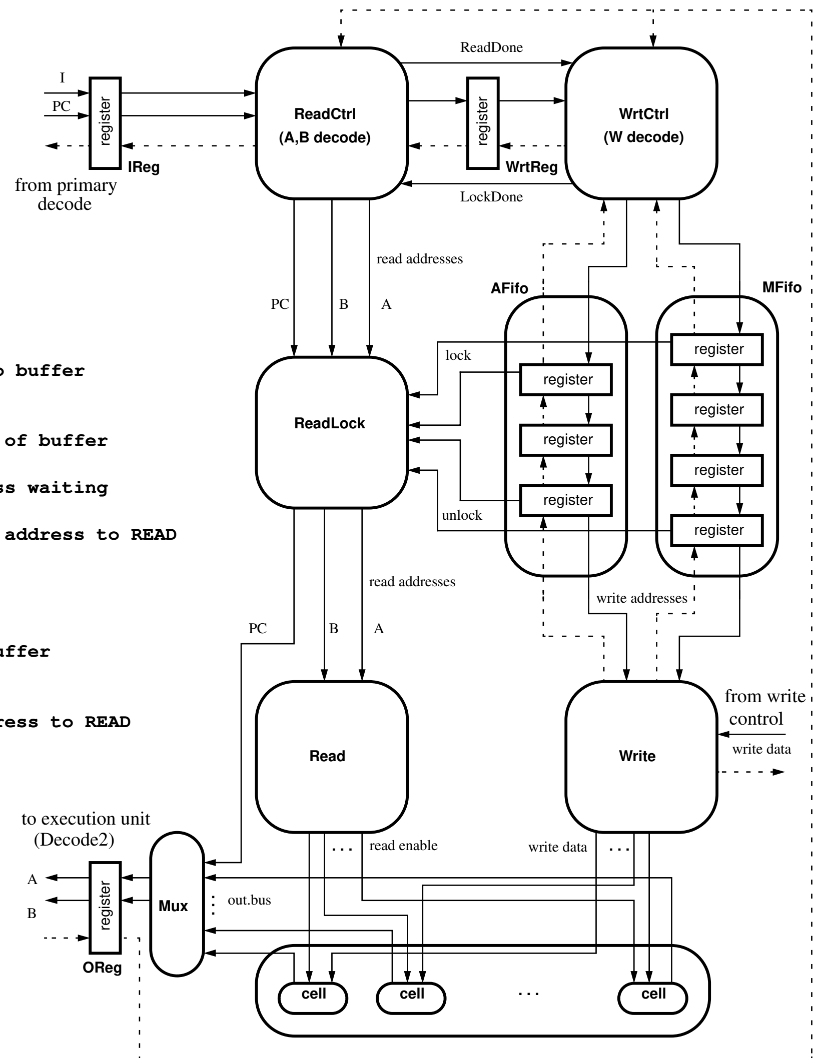


Figure 6.13: The Register Bank Model (RegBank)

a local copy of the locked addresses at any particular moment. The possession of this information, enables ReadLock to control the interaction between read and write operations and ensure that reads and writes take place asynchronously and concurrently when there is no interdependency.

The functionality of the ReadLock process is built around an ALT construct (figure 6.13). Read register addresses are sent to ReadLock from the ReadCtrl decoding process and are compared with the contents of the circular buffers. A match implies that the register is locked and thus the read operation is stalled until ReadLock is informed that the register has been removed from the lock FIFOs; no buffering is required in ReadLock for stalled read addresses as only one read operation is outstanding in the register bank at any particular moment. Read register addresses which are not locked, are forwarded to the *Read* process which, in turn, sends *read enable* messages to the register cell processes involved.

Register cells use an ALT to receive either read enable signals from *Read* or the new data to be stored from *Write*. A read enable signal causes a message with the register's contents to be output on the *out.bus* channel. For each instruction, at most two out.bus channels will be fired to provide input to the multiplexor process, which directs the messages to their appropriate destinations. The PC value is sent over a separate channel to ReadLock and it is either discarded or propagated to the multiplexor process to be output as an operand.

An alternative way to model the register bank would be to abandon the use of separate lock FIFO processes and replace both, the lock FIFOs and ReadLock by a single occam process. The functionality of this new process would be similar to that of ReadLock but with the circular buffers representing the actual lock FIFOs; a single ALT would handle all the interactions of the process with its environment including messages to be written to the registers.

Using this scheme, the implementation of the register locking is straightforward as the global view required is directly available by the circular buffers. However, having both read and write operations handled by a single process introduces a bottleneck which limits the concurrency of the model and degrades its performance; if a write operation was selected by the ALT, any outstanding read operation would have to stall and wait for the completion of the write even in the absence of any interdependency. The mechanism adopted allows read and write operations to be handled concurrently by different processes, a behaviour which increases the parallelism of the model and consequently its potential for

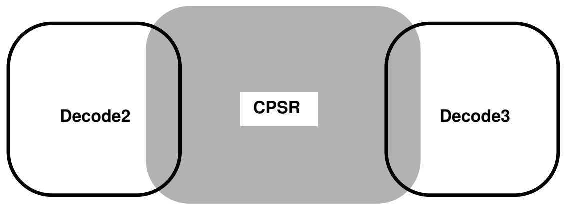


Figure 6.14: The Execution Unit of AMULET1

high performance.

6.8 The Execution Unit Model

As mentioned in section 4.6.2.4, the execution pipeline comprises two major stages, namely Decode2, which controls the operation of the shifter and multiplier units of the processor, and Decode3 which controls the ALU.

Decode2 and Decode3 are bridged by the CPSR (Current Processor Status Register) unit of the processor as depicted in figure 6.14. This unit incorporates the logic required for the calculation of the *Current Processor Status* (*CPS*), namely the arithmetic flags and the processor mode information. The hardware to evaluate the condition codes of executed instructions is also included in the CPSR unit.

6.8.1 The CPSR Model

The operation of the CPSR is based on information provided by both the stages of the execution unit (i.e. Decode2 and Decode3) as depicted in figure 6.15. The active CPS (flags and mode) is stored in *ALatch*. The condition evaluation hardware takes into account these values as well as the condition flags of the instruction (*Imd*, provided by Decode2, see section 6.8.2) and the operating colour of the processor (provided by the Decode3) to make a decision whether the instruction should be discarded as invalid. The active CPS is also used to

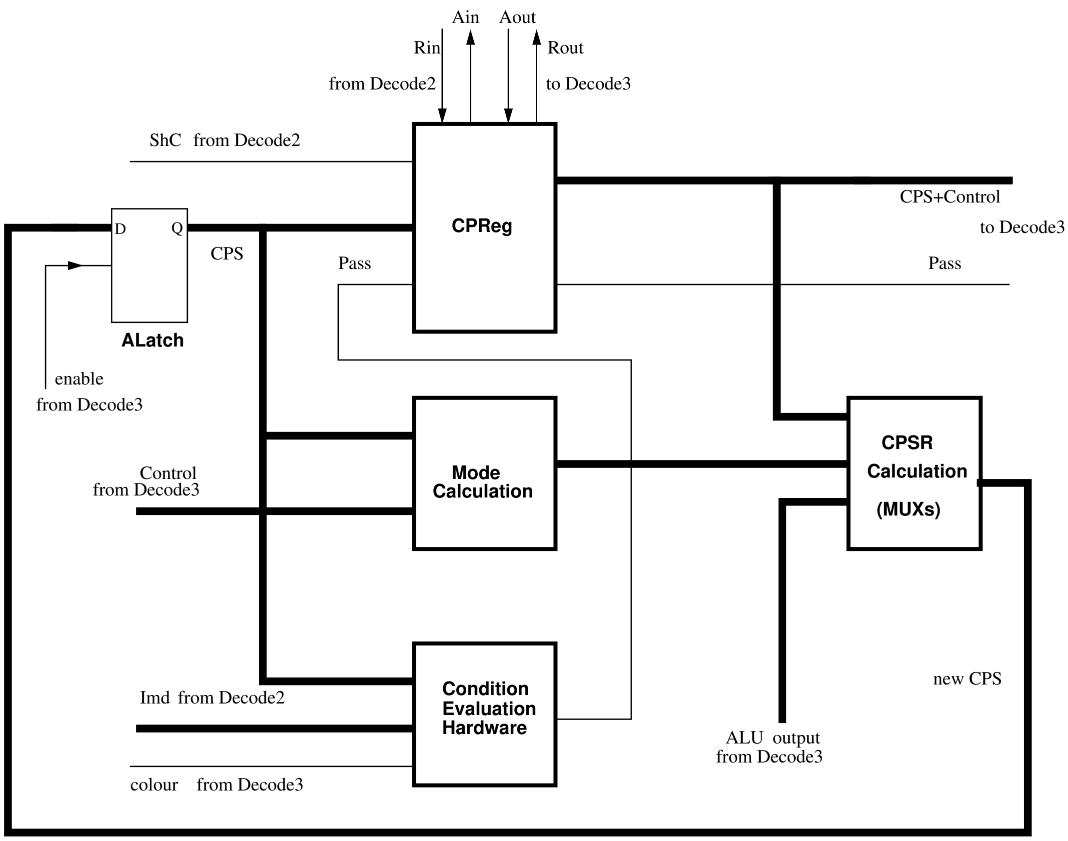


Figure 6.15: The CPSR Unit

calculate the current mode of the processor. The role of the *CPReg* register is to keep a copy of the CPS so that the arithmetic flags provided to the ALU remain stable while the new flags are being calculated.

The CPSR unit is another part of the AMULET1 where fully asynchronous operation was traded for efficiency and speed. The input of the CPReg register is controlled by Decode2 while the data latched by the register are provided by both Decode2 and Decode3. The timing characteristics of the execution unit allow the correct operation of the system. However, this situation where the input request to a register is issued by the previous stage in the pipeline, while the data of the same bundle is provided by the next stage, can not be described in occam, for it violates the handshake protocol that provides the basis for asynchronous micropipelined operation. An alternative implementation of the CPSR, which

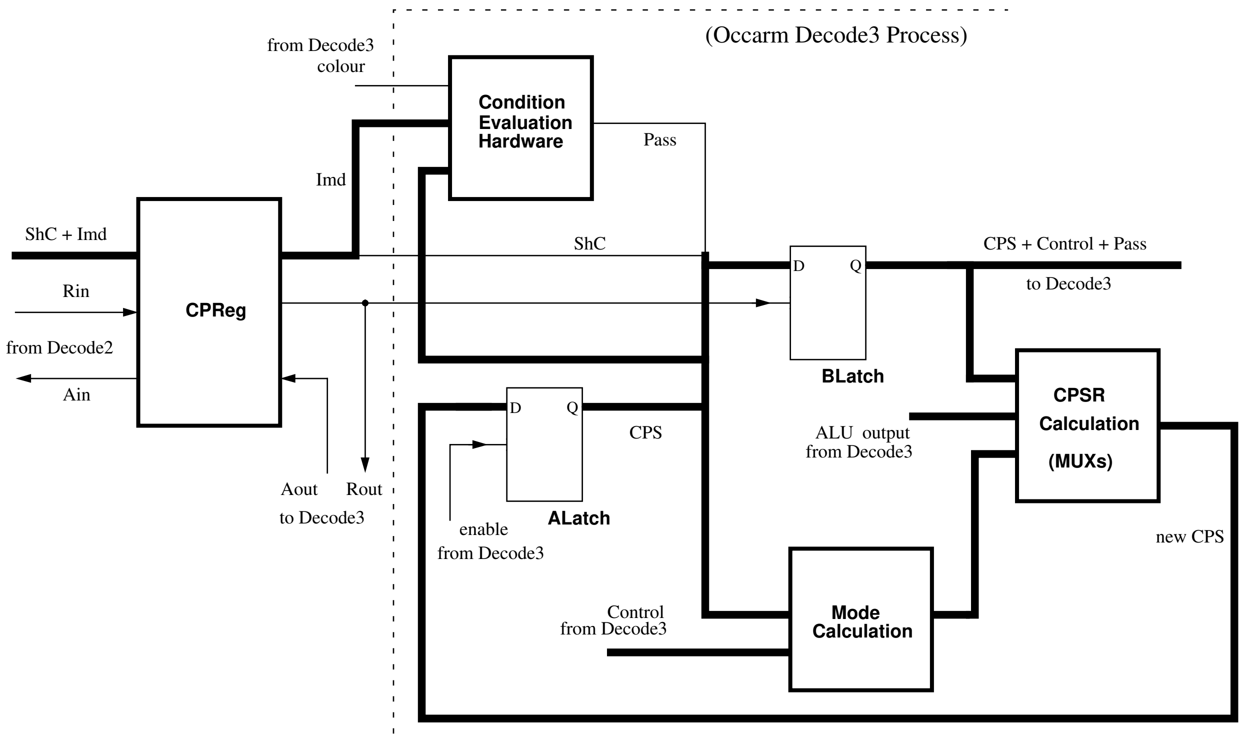


Figure 6.16: CPSR: An Alternative Design

emerged as a result of the effort to construct the occarm model, is depicted in figure 6.16. This implementation achieves a fully asynchronous and thus safer operation at the expense of an extra latch (the *BLatch*); this latch is enabled by the request event of the CPReg register to hold the copy of the CPS required to provide stable inputs to the ALU.

The handshake interfaces at the input and output sides of the CPReg register are clear and well defined making their modelling in occam straightforward. Within this implementation, the two latches, the logic for the calculation of the CPS and the condition evaluation hardware, logically form part of Decode3 since all the information required for these functions is available locally at this stage; in occarm, this circuitry has been modelled as a sequential piece of occam code in the Decode3 process.

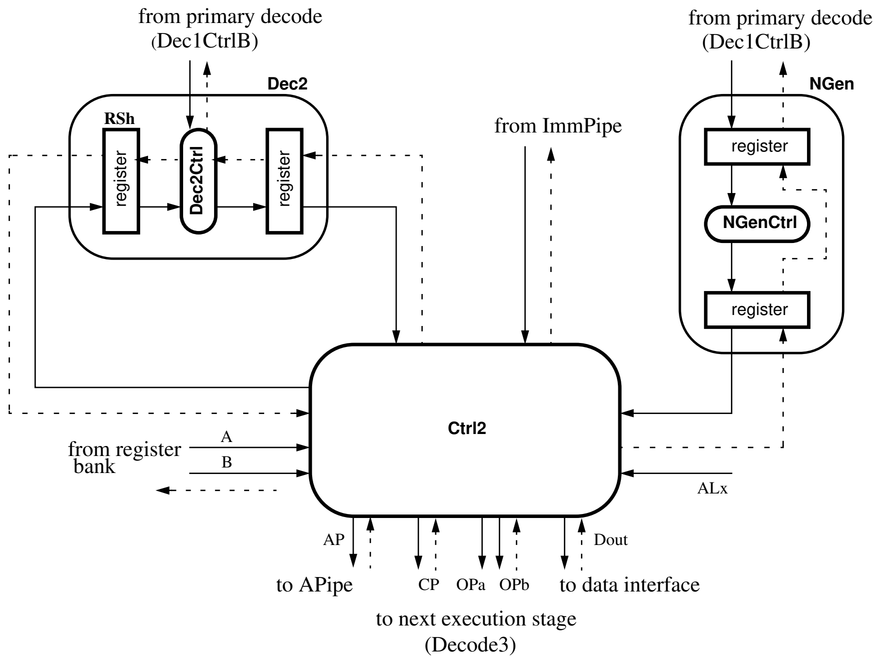


Figure 6.17: The First Execution Stage Model (Decode2)

6.8.2 Decode2

The process structure of the model of the first execution stage is depicted in figure 6.17. Control information generated by the partial decoding of instructions in the primary decode (Dec1CtrlB) arrives to the *Dec2Ctrl* process. Further decoding takes place at this process (using a PLA model) and the complete control information is forwarded to *Ctrl2* where it is paired with the operands of the instruction extracted from the register bank (A and B channels) or the Immediate Pipe. *NGenCtrl* process calculates the number of registers to be transferred in a load/store multiple operation from the bottom sixteen bits of the instruction; this value is used by the ALU to perform the required base calculations.

The multiplier and/or shifter are invoked as occam functions and the results of the operation are placed onto the OPa and OPb channels to be sent to Decode3. The carry required for the correct operation of the shifter is sent by the ALU via ALx channel.

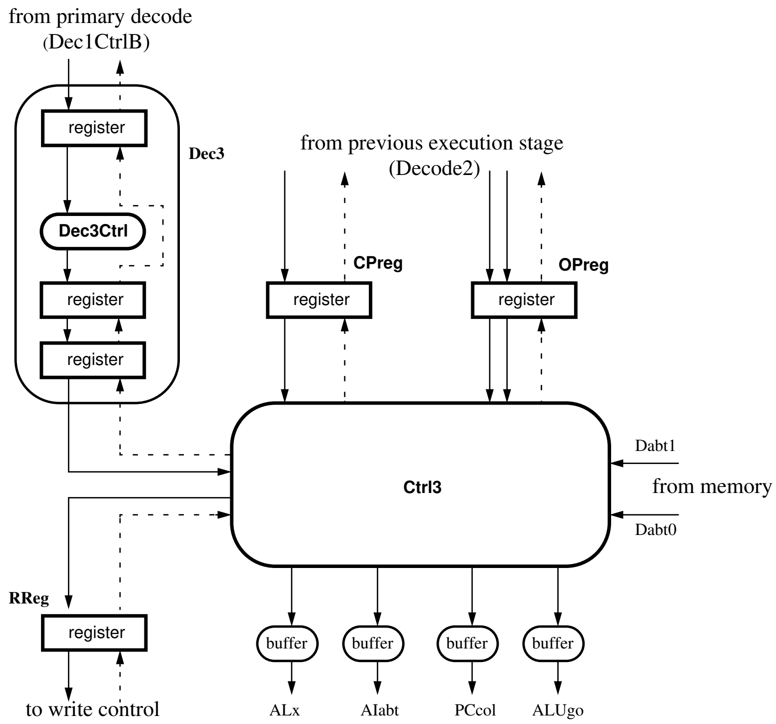


Figure 6.18: The Second Execution Stage Model (Decode3)

Data to be written to memory are sent intact to the data interface (to synchronize with their associated memory address from the address interface) via *Dout* channel. The *CP* channel provides input to the CPReg register described in the previous section. This information includes the carry produced by the shifter (*ShC*) and the condition flags of the instruction (Imd, generated and sent to Decode2 by primary decode). The *AP* channel provides input messages to the APipe in the address interface, while *Rsh* register temporarily stores the shift value during the first cycle of register based shift operations.

6.8.3 Decode3

Decode3 process is depicted in figure 6.18. *Dec3Ctrl* makes use of a PLA model to generate the control information required for the operation of the ALU. The PLA input is provided by the primary decode as a result of the initial partial

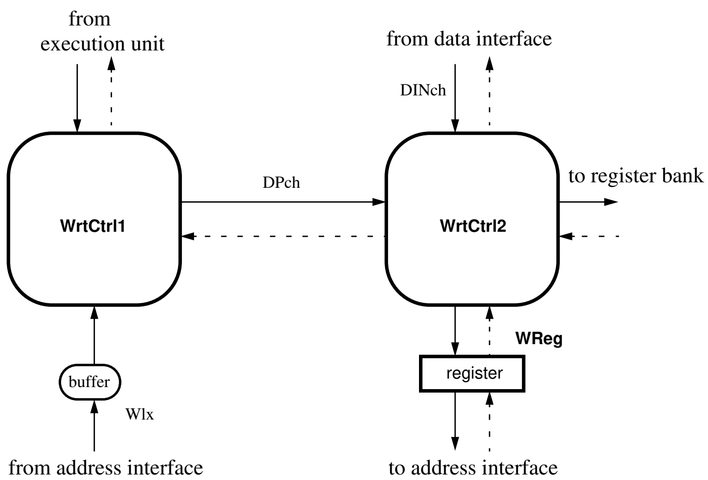


Figure 6.19: The Write Bus Control Model

instruction decoding. *Ctrl3* is used to invoke the ALU; it also incorporates the CPSR logic of figure 6.16.

The input operands of the ALU arrive from Decode2 through *OPReg* while the results of the ALU operation are forwarded via *RReg* to the register bank (write data) and/or the address interface (data address or new PC). *ALUgo* and *PCcol* channels are used to send back to the primary decode the current mode (Dec1CtrlB) and colour (Dec1CtrlA) of the processor respectively. Abort signals from memory arrive on Dabt0 and Dabt1 channels while AIabt is fired each time a data transfer instruction is invalidated and discarded (see section 6.5.3.4).

6.9 The Write Bus Control

WrtCtrl process (figure 6.19) models the arbitration logic which provides access to the *write bus* of AMULET1. The arbitration is performed in the *WrtCtrl2* process by means of an occam ALT construct on messages arriving from memory (via *DataIn* process in data interface) and the execution unit (*Decode3* via *WReg*); these messages are directed either to the register bank (write data) or the address interface via *WReg* register (data addresses arriving from the execution unit or new PC values sent from either the execution unit or the memory).

The role of the *Wlx* signal is to synchronize the address interface with the write bus control logic in order to avoid deadlock situations which might occur as a result of the write bus getting locked while Wreg is full waiting for an acknowledgement from the address interface. A new message arriving from the execution unit will not be forwarded until the address interface issues Wlx to indicate that Wreg is free.

6.10 Summary

This chapter has described occarm, a model of the AMULET1 asynchronous microprocessor developed using the proposed modelling philosophy and employing occam as a description language.

The next chapter discusses issues related to the execution of the occarm model, and occam models of asynchronous architectures in general, on a multi-transputer system.

Chapter 7

Simulation Issues

7.1 Introduction

In a sequential environment, the execution and testing of a simulation model written in a conventional sequential language is a straightforward issue.

This however does not apply to asynchronous parallel models. The extra dimension that concurrency introduces to distributed systems [Back78] complicates the programming activity and imposes a number of issues which need to be addressed before any parallel program may be executed. These issues include monitoring, debugging and terminating the processes of the system; for multiprocessor configurations, remote I/O, partitioning, mapping and load balancing issues need also to be addressed. This is particularly true for the development of occam systems, since the static nature of occam and the lack of a transputer operating system make the aforementioned tasks the responsibility of the application developer; recent versions of the occam toolset, however, provide Virtual Channel support (at added runtime costs).

This chapter discusses how the above issues have been addressed in the context of the occarm simulation model. Two different environments for occarm have been developed, one for the execution of the model on a single transputer and

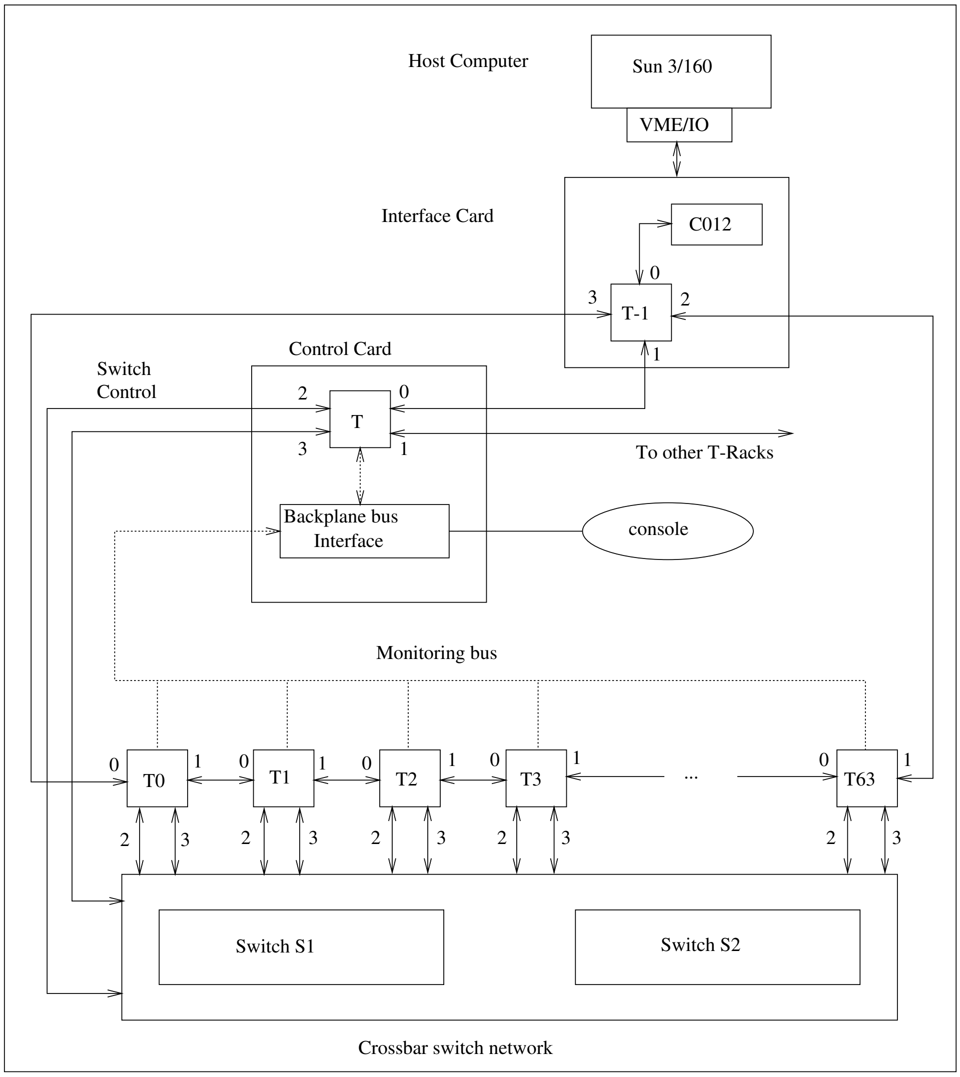


Figure 7.1: The T-Rack

one which provides for a multi-transputer implementation.

7.2 The Host Machine: The ParSiFal T-Rack

The computer which hosts the occarm simulation model is the ParSiFal T-Rack [Know87], a reconfigurable, transputer based machine which has been developed at the University of Manchester as part of the *Parallel Simulation Facility* project

[Capo86] under the U.K Alvey program[1] [Logi85]. The T-Rack was developed to serve as a facility for the parallel simulation of computer architectures.

The basic architecture of the T-Rack is illustrated in figure 7.1. It comprises sixty four T800 transputers (*T0-T63*), each with one or two Megabytes of local dynamic memory. The transputers are housed on sixteen identical boards (four transputers per board). Each transputer communicates via four asynchronous bidirectional links numbered from 0 to 3. Two of the four links from each transputer of the T-Rack (*link0* and *link1*) are permanently hardwired to form a processor chain known as the *necklace*.

The off-necklace links (*link2* and *link3*) may be connected by means of a crossbar switch which is built using twenty six INMOS C004 switch chips housed on two boards (*S1* and *S2*, also referred to as *near* and *far* boards respectively [Gars89] [Murt91]). Connections within the switch networks may be defined using a software switch utility allowing the transputers to be connected to form the configuration required [Jone87]. The switches are statically set before the application is loaded, though dynamic reconfiguration is also possible [Jone88] [Murt91]. The crossbar switches also provide for the connection of transputer links to external devices.

The T-Rack is hosted by a Sun 3/160 workstation [Sun86] containing a *Tadpole Transputer Board* [Tadp87] which acts as a "root" node for the T-Rack and is used for the downloading of code on to the rack and for I/O operations to and from the host machine. The Tadpole transputer forms part of the necklace (T-1, see figure 7.1).

The control board is used for switch control functions and for system back-plane monitoring which is achieved by means of a byte-wide monitoring bus. This bus provides an alternative route between the T-Rack and the outside world, as a

[1] The ParSiFal industrial and academic partners also included Logica plc., Inmos Limited, GEC Research Limited, FEGS Limited, The Polytechnic of Central London and the Engineering Department of Cambridge University.

terminal may be attached to the control board to display monitoring information [Know89]; the occarm simulator does not make use of this bus.

The existence of the necklace in the T-Rack limits the set of processor graphs which may be implemented to those which possess a Hamiltonian Cycle [Murt87]. An occam program can execute on the T-Rack if the required processor network, either contains, or can be modified to contain a Hamiltonian Cycle. The route taken by the Hamiltonian Cycle through the network corresponds to the necklace of the T-Rack while the edges not on the Hamiltonian Cycle are mapped on switched links.

The switching network implements a "split-link" switching policy, whereby link 2 outputs are connected to link 3 inputs via one switch board, while the connection between link 2 inputs and link 3 outputs is achieved via the other switch card[2] [Hill86].

7.3 Monitoring

Monitoring the runtime behaviour of the simulation model and collecting information regarding the characteristics of the simulated system is one of the main objectives of the simulation process. Monitoring is essential for the testing and performance evaluation of the simulated system as well as for the debugging of both the simulated system and the simulation model.

The inherent properties of distributed asynchronous systems make monitoring a difficult and complicated issue for which sequential techniques are insufficient [Riek94]; distributed monitoring is currently an active area of research.

The main problems which are associated with distributed systems and which an ideal monitoring system should address include the multiple threads of control,

[2]The split link switching mechanism provides a solution to the *Odd Cycle problem* which does not allow the construction of networks that possess an off-necklace closed path (cycle) which has an odd number of edges [Murt87].

the intrusiveness, the non-determinism and the need to cope with a vast amount of monitoring data [Riek94] [Joyc87]:

The Multiple Threads of Control

To understand of the system's behaviour, it is essential to be able to get a global view of the system (a snapshot) at any particular moment. In sequential programming this is straightforward. In distributed simulations however snapshots are not easily obtainable; to determine the system state, all the different local process as well as channel states need to be taken into account [Chan85] [Baba93]. The monitoring system should be able to correlate the histories of the different processes and put them in a global temporal perspective. This is a complicated issue related to the problem of dealing with causality in a distributed environment, discussed in section 3.4.

An approach for constructing snapshots of a distributed system, which is employed by most of the existing monitoring tools, is to collect runtime traces of appropriate event records and put them in a temporal perspective off line, in a postmortem way.

Non-Determinism

Distributed asynchronous systems are nondeterministic structures. Multiple executions of the same system may produce different, albeit valid, ordering of events. This makes the repeatability of an error situation a difficult task, although systems which achieve a postmortem recreation of the system's state based on a transcript file of event records have been developed [Garc84] [Unge86].

Intrusiveness

The monitoring system should not mask or affect in any way the behaviour and characteristics of the simulation model. Hardware monitoring tools are usually non-intrusive [Riek94]. However, using a software environment for monitoring alters the behaviour of the distributed system [Mari92]. Indeed, monitoring facilities make use of the host machine's resources, thus imposing delays in the simulator's execution; the host machine executes alternately simulation and monitoring code (e.g. in a time-sliced fashion) and the monitoring messages produced share the communication network. As already explained in section 5.3.1.1, delaying a process alters the event timings in the distributed system in an arbitrary way, thus changing its behaviour[3]. Clearly, software monitoring also has an impact on the distributed simulation's performance.

Although the intrusiveness of software monitoring is recognised, hitherto no precise model has been developed for the quantification of these effects [Riek94]; generally efforts are focused in finding ways to minimize the influence of the monitoring on the simulation system (e.g. [Sega86]).

Dealing with Monitoring Data

This involves the generation, transportation, representation and analysis of event records.

Monitoring messages may be generated using either a time driven or an event driven approach [McLa92]; in the former, the monitoring system scans, at regular intervals, event records from the simulation model (time sampling), while in the latter the simulation model detects and reports to the monitoring system event records on its own initiative. In software monitoring, the detection of events to be reported is achieved by means of *probes*, small pieces of code which are inserted

[3] In a sequential environment, a program can be stopped and resumed without the elapsed time between the two actions affecting its behaviour.

into the simulation model's code triggering the generation of monitoring messages each time they are executed[4] (this process is referred to as *instrumentation of the simulation model*). Typically, probes are inserted in the model's source code[5] manually, though some monitoring tools exist which attempt this automatically [Mohr90] [Lump92].

Typically[6], in distributed simulation, monitoring information needs to be transported for processing to a remote node of the distributed system; this is particularly true for transputer based systems as only the root transputer of the network has access to the file system of the host machine. The monitoring data are generated in a copious volume and their transportation can have a significant negative impact on both the computational resources and the communication network of the distributed system. To address this problem several *transport strategies* have been developed, the most important of which are *immediate transport*, which sends the monitoring data as soon as they are generated (e.g. [Bemm90] [Leu92]) and *store and unload*, whereby the data are stored in a buffer before they are transported; in the latter, the buffer may flush its contents when it becomes full, upon request, when the communication load is low or when the simulation is complete (*store and unload afterwards*) [Riek94].

Once the monitoring messages have been collected, they must be analysed to extract the information which will enable the user to draw conclusions regarding the behaviour of the simulated system. This is not an easy task as the monitoring

[4]A probe is always inserted next to (before or after) the instruction it monitors.

[5]This process is referred to as *source code instrumentation* as opposed to *object code instrumentation* wherein the probes are inserted into the object code at compile time by an *instrumenting compiler*; schemes which have the probes inserted in the operating system (*instrumenting kernel*) also exist for monitoring the activities of the simulation model which invoke the kernel functions [Malo90] [Rudo89] [Bemm90] [McLa92].

[6]Quoting Riek "no entirely distributed monitoring tool has yet been developed (i.e. a complete parallel program that will gather and use the monitored information). In the existing monitors, the runtime information is used at a different place from where it was generated" [Riek94].

Process **process_name:** instruction **instruction_type** was executed at time **clock**

a) Execution event trace

Process **process_name:** variable **variable_name** has the value **val** at time **clock**

b) Data event trace

Process **process_name:** sending value **val** to process **proc** on channel **chan** at time **clock**

Process **process_name:** receiving value **val** from process **proc** on channel **chan** at time **clock**

c) Parallelity event traces

Figure 7.2: Event Traces for Debugging

data represents an enormous amount of information[7], usually at a fairly low level of abstraction, which also encapsulates a high degree of parallelism implicitly contained in the event records; representing the monitoring data in an appropriate way is crucial to enable to user deal with it. This is an active area of research related to the Human Computer Interaction field. In existing tools the monitoring data is presented to the user in the form of textual traces and pictures (graphical visualizations) although tools are currently under development which employ sound too [Zaba92].

7.3.1 Monitoring Occarm

Occam architectural models of micropipelined systems are distributed asynchronous structures and consequently their monitoring imposes the problems described in the previous section.

The simulation of an asynchronous architecture has two main objectives:

1. The testing and debugging of the architecture.

2. The evaluation of the architecture's performance.

[7]Different techniques such as *event record filtering* [Holl93], *event record clustering* [Mohr90] or *event abstraction* [Bast94] have been proposed to reduce the amount of monitoring information in distributed systems.

7.3.1.1 Debugging

For the debugging of the architecture (as well as the simulation model) it is necessary to monitor both the flow of control and the flow of data in each of the different occam processes in the model; this is achieved by collecting traces regarding the *execution* and *data events* of the processes respectively [Riek94] (figure 7.2a,b).

For the detection of deadlocks, it is essential to know the state of the channels in the system when the deadlock occurred. For this purpose, the *parallelity events*, which correspond to communication actions, need to be monitored (figure 7.2c). These will appear in pairs, one for the sending and one for the receiving process. The probe in the sending process code, is inserted before the output command while the receiving probe follows the input command. The absence of one parallelity event from a pair in the final trace indicates the occurrence of a deadlock.

In occarm, monitoring is performed by means of a parallel network of occam monitoring processes (known as *reactive processes* [Mari92]), one for each of the top level processes (the *active processes*) of the occarm model[8].

Monitoring messages are issued to the reactive processes in an event driven fashion. Probes, in the form of a procedure call, have been inserted (manually) in the source occam code of the active processes. Each time the procedure which implements the probe is called, it constructs the corresponding monitoring message and sends it to the monitoring process. Since probes will be invoked in different parallel sections of the active process, several monitoring messages are issued simultaneously. Thus, for the communication between an active and the corresponding reactive process, a channel array is used (*monitor*, see figure 7.3);

[8]Within the ParSiFal project, a number of experimental graphical tools were developed which illustrate or monitor different aspects of an occam parallel program [Step86] [Step88]; these however were not used in connection with occarm due to their limited capabilities and the need for portability of the occarm simulation environment.

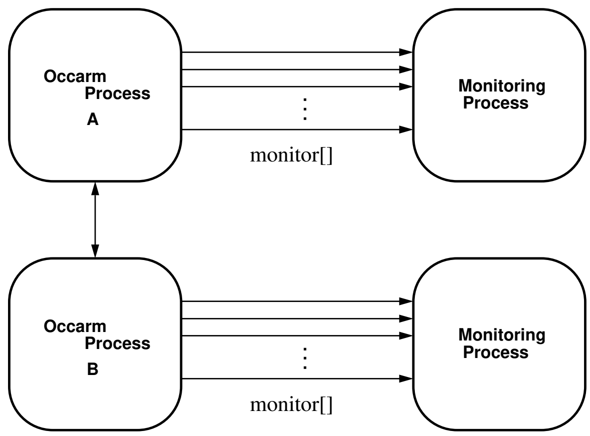


Figure 7.3: Collecting Event Traces in Occarm

for the current implementation of occarm the total size of the monitoring channel array is fifty (50).

The monitoring process acts as a multiplexor, employing an ALT construct to gather the messages issued by the corresponding active process on the channel array; in order to reduce the effects of the ALT bottleneck, the channel array is buffered to decouple the processes involved.

Monitoring processes support both immediate and store and unload transport policies. The latter makes use of a circular buffer to store only the recent history of the active process. If no monitoring message arrives for a user-defined time interval (i.e. in the case of deadlock), the monitoring process flushes the contents of the history buffer.

The store and unload transport option is particularly useful as in most, if not all, cases the most recent history of the processes is sufficient to identify the cause of errors or deadlocks. Since the monitoring messages do not propagate further into the system, the communication overhead of the store and unload policy is minimal (see section 8.4).

7.3.1.2 Performance Evaluation

When analysing the performance of asynchronous pipelines, measures of special importance and interest are the *occupancy* as well as the *stall* and *idle* periods of the pipeline.

The occupancy of a N stage pipeline is defined as the percentage of time the pipeline has 1, 2, ..., N elements in it [Furb94c].

In a synchronous pipeline, the clock frequency defines the period that any element stays in the pipeline; thus for the calculation of the occupancy only the entry (arrival) times of elements are required.

In asynchronous micropipelines however, the times that a particular data bundle enters and leaves the pipeline are arbitrary. Therefore, the calculation of occupancy in this case requires knowledge of both, the entry and exit times of bundles in the pipeline. A bundle enters a pipeline when the corresponding data is latched by the first register of the pipeline; thus the entry time is represented by the timestamp of the acknowledgement (Ain) signal issued by this register. Similarly, the exit time of a bundle is the value of the timestamp of the Acknowledgment signal to the last register of the pipeline.

The values of the two timestamps required for the calculation of the duration of a message's staying in the pipeline are not directly available, for they are possessed by different occam processes and occam does not support global variables. To overcome this problem a solution has been devised whereby request messages exiting the pipeline carry with them an extra timestamp denoting the time of their entry. Using this information, the calculation of the pipeline occupancy by the control process at the output side is straightforward.

In occarm, control processes maintain a set of *occupancy tables*, one for each of their input pipelines. The occupancy table is a circular buffer which contains the input and output timestamps of messages passing through the corresponding pipeline, thus providing a (postmortem) global view of the pipeline at any particular moment.

Each time the control process at the output side issues an acknowledgement message to a pipeline (i.e. each time a message exits the pipeline), it also invokes a probe procedure (the *calculate.occupancy()*) to calculate the current occupancy values for the pipeline; idle periods are also calculated at that point by the probe.

Contrary to the debugging traces which are sent immediately to the corresponding monitoring process, the type and quantity of monitoring values concerning the performance characteristics of the architecture permit active occarm processes to calculate and store them locally; this eliminates the extra communication overhead that their transport would impose. The stored values are unloaded by the occarm control processes upon their receiving the termination signal (see section 7.4).

Stalls. An asynchronous pipeline will stall if the rate that request events are issued to the pipeline is greater than the rate that events propagate through the pipeline or the rate that events exit the pipeline (i.e. the rate that events are consumed and processed at the output side).

Stall situations refer to the input side of the pipeline. They may be detected by examining the delay between the sending of a Request event (Rin) to the pipeline and the issuing of the corresponding acknowledgement signal (Ain) by the first register of the pipeline: a stall situation has occurred if $timestamp(Rin) < timestamp(Ain)$. The duration of the stall is $timestamp(Ain) - timestamp(Rin)$; clearly the minimum stall period is equal to the propagation delay of the first register in the pipeline.

Monitoring information regarding pipeline stalls is collected by the occarm

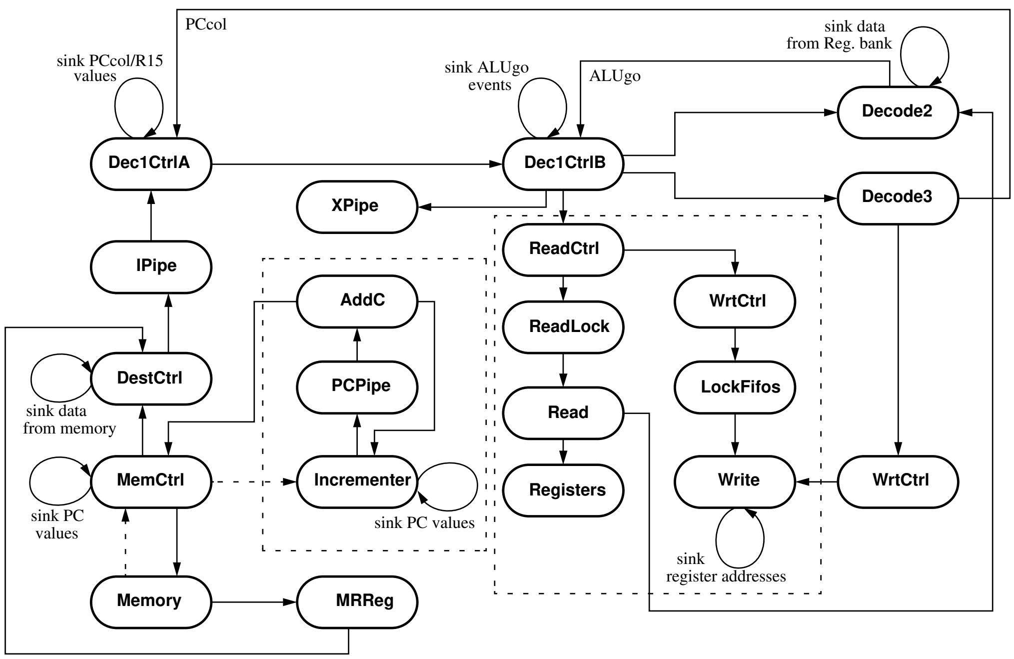


Figure 7.4: Terminating Occarm

control processes at the input side and is unloaded upon detection of the termination signal.

7.4 Termination

Detection of termination in a distributed parallel environment was brought to prominence in 1980 by Francez [Franc80] and by Dijkstra and Scholten [Dijk80] and since then has constituted one of the basic problems in distributed computing. The fundamental problem in detecting termination is the difficulty of constructing a global state of the distributed system (see section 7.3). Several termination detection algorithms have been developed; these differ in the way they ensure correctness[9] and the assumptions they make about the semantics and behaviour

[9] The detection algorithm has to report termination in finite time (liveness); if such a report occurs, then the underlying distributed system must have indeed terminated (safety) [Brze93].

of the communication links in the distributed system[10]. Typically, termination mechanisms consist of a *termination detector*, superimposed on the distributed system, which either monitors the activity of the processes or uses a deadlock detection/breaking approach [Brze93].

A simulation model of an asynchronous architecture has completed its operation, and thus must terminate, if it has executed all the instructions of a particular benchmark program. Within the ARM development environment used in the AMULET project [ARM], ARM programs notify their completion by writing a special End Of Program (EOP) character to a particular address in memory.

The above mechanism may be exploited for the termination of the occarm model too; upon receiving EOP, the memory process issues a *KILL* message which then propagates through the model, progressively killing the occam processes. A possible route for the KILL message which has been adopted for the termination of occarm is depicted in figure 7.4. The KILL signal enters occarm by means of the Acknowledgment message issued to the MemCtrl process by the memory for the EOP, and is then forwarded to Dec1CtrlA (through the IPipe) and to the Incrementer (on the corresponding Acknowledgment message) to terminate the datapath and address interface respectively. To cope with closed paths (loops) in the model and allow the KILL message to reach all processes in the loop, certain processes forward the KILL signal but do not terminate immediately; they continue their operation sinking subsequent messages until they receive the KILL for a second time. These processes include MemCtrl and the Incrementer, which sink PC values from the PC loop, DestCtrl, which sinks messages sent from memory before EOP, Dec1CtrlA, for PCcol signals and prefetched instructions, Dec1CtrlB, for ALUgo signals, Decode2, which sinks data from the register bank, and the Write process in the register bank, which sinks register addresses arriving

[10]These assumptions distinguish between synchronous or asynchronous communication, FIFO or not, atomicity of communication actions etc.

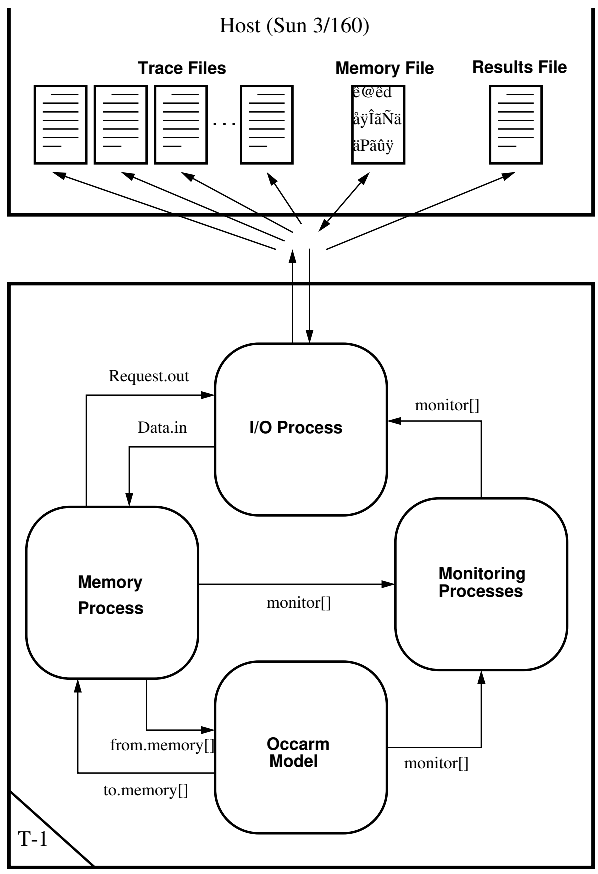


Figure 7.5: The Single transputer Environment of Occarm

from the lock fifos.

7.5 The Simulator Environment

The single-transputer simulator environment is depicted in figure 7.5.

The Memory process models the memory control logic of the processor; the memory itself is implemented as a binary file to achieve compatibility with the existing ARM development environment [ARM]. The operation of the simulator consists of reading instructions from the memory file, executing them and, possibly, writing results back into it; for compatibility reasons, messages regarding the correct operation of the simulated architecture produced by benchmark programs, are written to a separate text file, one character at a time.

Within the INMOS occam toolset environment, only one occam process may have access to the host machine's file system [Inmo91a]. In the occarm simulator environment this role is served by the I/O process via which, all interactions with the outside world are performed. The I/O process employs an ALT construct to allow the multiplexing of system and monitoring messages arriving from the Memory and Monitoring processes respectively.

Event traces arriving from the monitoring processes are distributed to different trace files according to their "process_name" field (see figure 7.2); the existence of a separate trace file for each process provides a view of the parallelism of the system and thus facilitates the postmortem debugging task.

7.6 Multiprocessor Implementation

As discussed in section 5.3.1, one of the advantages of using occam as a specification language for asynchronous architectures, is the ability to exploit the inherent parallelism of the simulated architecture, and thus to achieve higher performance, by executing the simulation model on a multiprocessor machine.

In order to exploit this potential, a multi-transputer configuration of occarm on the T-Rack has been developed.

7.6.1 Mapping Occarm onto the T-Rack

Mapping a parallel program onto a parallel machine[11] is one of the fundamental and most difficult problems in parallel processing[12] and a detailed discussion of

[11]The mapping problem is essentially a graph embedding problem, and may be described as follows: Given two graphs $G1$ (the program) and $G2$ (the multiprocessor machine), find a function $f : G1 \rightarrow G2$, such that certain performance criteria are satisfied.

[12]Actually, the mapping of a parallel program onto a parallel machine is only a particular manifestation of the more general mapping problem which stems from the very nature of

the mapping problem would exceed the scope of this thesis. The excellent paper by Norman and Thanisch [Norm93] provides a comprehensive list of references concerning the subject.

The mapping problem has been investigated within the context of parallel simulation too, for both conservative [Nand92] [Bouk94] and optimistic approaches [Reih90a] [Glaz92].

The static nature of the occam language requires that the mapping of the occam process graph on the transputer network is specified in advance by the application developer, though a number of tools have been developed to automate various steps of this task [Boil87] [Murt87] [Lau88] [Theo91] [Theo94c]. The mapping scheme should ideally take into account the following considerations[13] [Murt91]:

- The limitations imposed by the four links of the transputer: each node of the top level process graph should have at most four neighbours[14].
- The limitations imposed by the interconnection network of the machine. Typically, transputer networks are not fully connected and therefore process graphs have to be transformed to match the underlying structure; the aim here is proximity, namely placing communicating processes as close to each other in the network as possible.
- The computation and communication load should be evenly balanced over the transputers and the links of the system respectively.

Computer Science: Computer Science deals not with the objects themselves but, rather, with the *representation* of objects. This representation involves the reconciliation of conflicts between the *logical structure* of objects and the *physical medium* wherein the objects are dealt with; this accommodation takes the form of mapping problems [Rose94].

[13] As discussed in section 2.6.2.2, the T9000 transputer with the message forwarding facilities of the associated C104 link switch device eliminates most of these considerations; however for systems which are based on older generation transputers (this covers the vast majority of existing transputer based machines, including the ParSiFal T-Rack) these considerations have to be taken into account.

[14] More formally, the degree of each node in the graph should not be greater that four, with an edge in the graph representing a bidirectional link.

	Decode1	Decode2	Decode3	DatInt	AddInt	RegBank	WrtCtrl	Memory	I/O.Process
Decode1		*	*	*	*	*			
Decode2	*		*	*	*	*			
Decode3	*	*			*		*	*	
DatInt	*	*			*		*	*	
AddInt	*	*	*	*			*	*	
RegBank	*	*					*		
WrtCtrl			*	*	*	*			
Memory			*	*	*				*
I/O.Process								*	

: Merge into one link

Figure 7.6: Occarm Process Connectivity Table

Occarm Process	Node
Decode1 + Decode2	1
AddInt	2
Decode3 + DatInt	3
RegBank	4
WrtCtrl	5
Memory	6
I/O.Process	-1

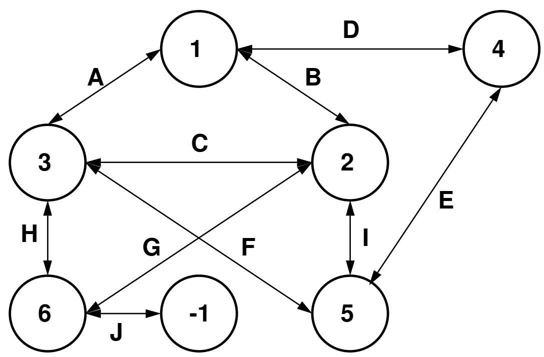


Figure 7.7: Modified Occarm Top Level Process Graph

Based on the above considerations, the first step for mapping occarm onto the T-Rack is the modification of the the top level process graph of occarm as depicted in figure 6.1 so that each node has at most four neighbours.

To address this issue the Process Interconnection Table (*PIT*) depicted in figure 7.6 has been devised. The number of asterisks in a row of the table represents the number of neighbour processes of that particular process. Merging two columns together, effectively adds one more level of abstraction to the process hierarchy, assigning the corresponding processes to the same processor and forcing the two channels to share the same link.

Link	No. of Multiplexed Channels
A	13
B	9
C	4
D	6
E	3
F	6
G	2
H	8
I	3

Table 7.1: Communication Load on Occarm Links

7.6.1.1 Balancing the Workload

The criterion adopted for the selection of the level of the occarm process hierarchy, each of whose process has at most four neighbours, is the maximization of processor utilization; namely, to occupy as many processors as possible.

Following this criterion the merges presented in figure 7.6 have been applied to occarm, deriving as a result the alternative graph of figure 7.7; this graph represents the[15] lowest level in the process hierarchy which satisfies the four-link-per-transputer limitation.

7.6.1.2 Balancing the Communication Load

The new top level occarm process graph possesses more than one Hamiltonean cycle, thus allowing an equivalent number of possible mappings on the T-Rack.

For the selection of the appropriate mapping, the criterion which has been followed is to balance the communication load. In the T-Rack the communication performance of a hardwired link is approximately double that of a switched link[16].

---

[15] A possible alternative would be to merge columns 4 and 5, and 7 and 8 (placing onto the same processors DatInt/AddInt and WrtCtrl/Memory respectively), instead of columns 3 and 4. This arrangement would require five, instead of seven, processors.

[16] The performance of a hardwired unidirectional link is reported as being 1.72 Mbytes per second, with that of a switched link being 0.87 Mbytes per second [Murt91], pp 63-65; this is

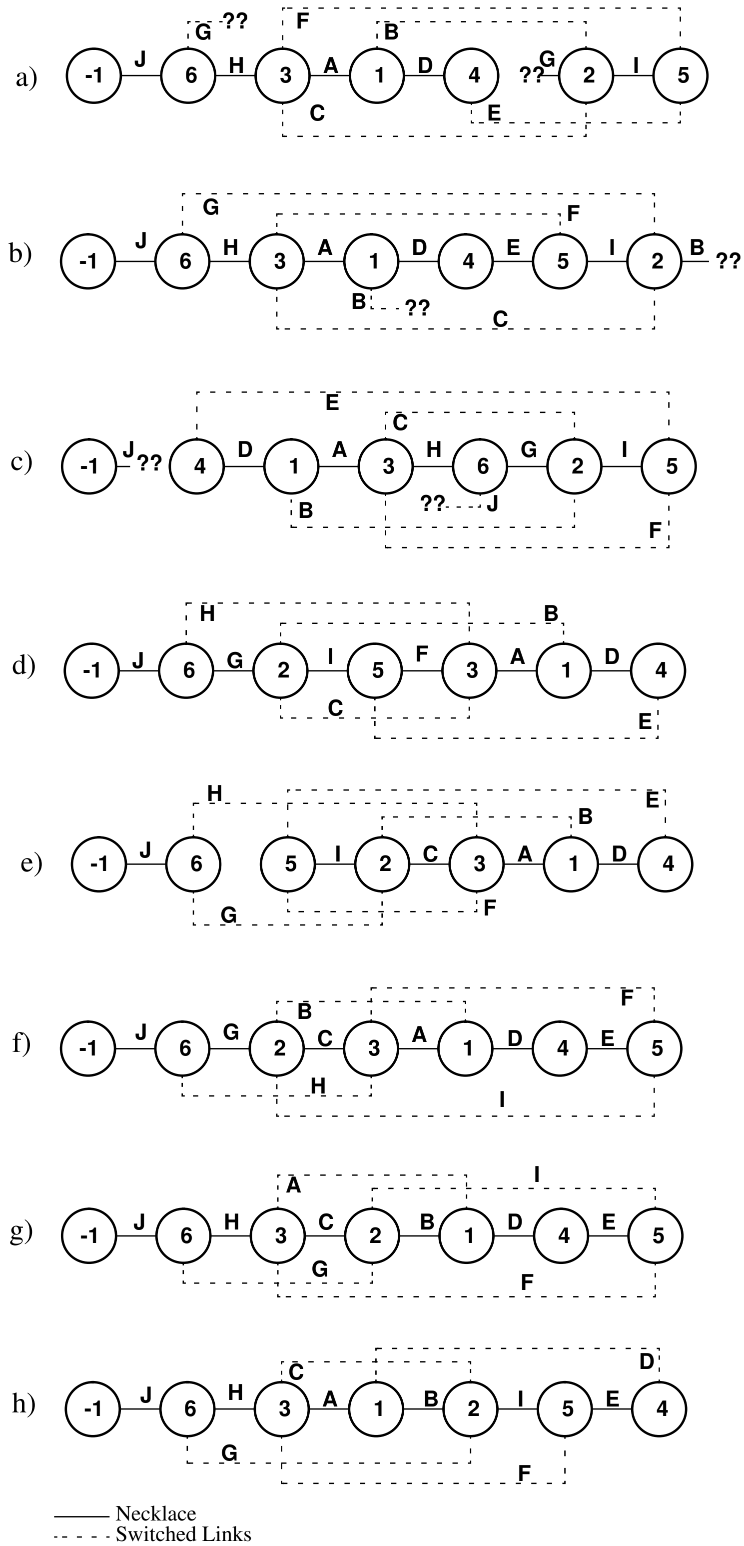


Figure 7.8: Occarm Graph Mappings

due to the increased latency of acknowledgment messages imposed by the C004 link switches.

Consequently, the objectives of the communication load balancing policy are to:

- Use as few switched links as possible, and
- Place onto hardwired links as many (multiplexed) channels as possible.

The latter is based on the assumption that on average, during the execution of benchmark programs, all channels in the system will have similar traffic levels. The pipelined structure of asynchronous architectures provides a basis to this assumption; indeed instructions, as they execute, propagate through successive stages of the pipeline thus activating most, if not all, channels on their path. Thus in the initial stages of the design process, when no data regarding the behaviour of the simulated architecture is available (as was the case with the AMULET1, when occarm was developed), the above mapping criterion provides a reasonable option. Once a detailed performance analysis of the architecture has been performed, the mapping of the simulator may be altered accordingly; in this case, a possible criterion would be to map onto the fast links as many as possible of the channels which are part of the architecture's critical path.

The communication load on the links of the top level occarm process graph is given in table 7.1. Figure 7.8 presents alternative mappings of the graph onto the T-Rack; $a \ldots c$ are examples of mappings which are not feasible due to the limited link connectivity of the T-Rack. The remaining mappings, namely $d \ldots h$, illustrate the different feasible ways to map occarm onto the T-Rack. From these, mapping h satisfies the communication balancing criteria specified above and therefore has been selected for occarm. This mapping uses the minimum possible number of switched links (namely 4) while allocating the maximum total number of channels onto the necklace (namely 36); mapping e makes use of 5 switched links while mappings d, f and g place onto the necklace 30, 28 and 30 channels respectively.

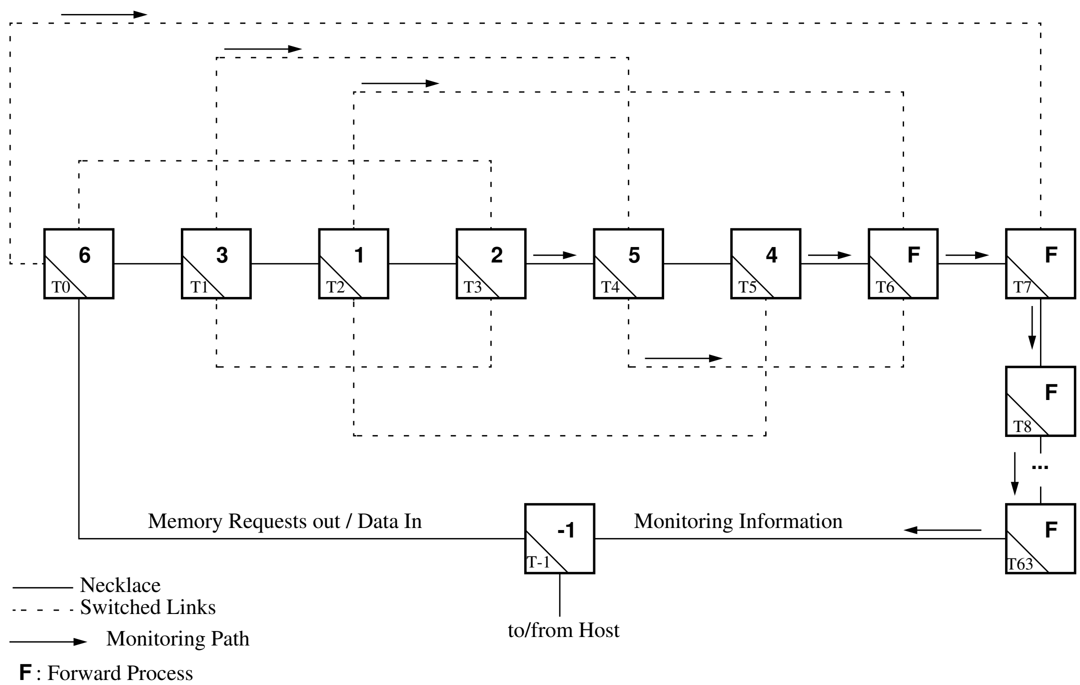


Figure 7.9: Mapping Occarm onto the T-Rack

The selected mapping h, has also the advantage that the maximum number of channels from the Decode3-Memory path (namely links F, H and I) are placed on the necklace. This is the path followed by the data transfer addresses and the corresponding abort signals during the execution of data transfer instructions. As explained in section 6.5.3.4, after sending the data transfer address to memory, Decode3 blocks until the corresponding abort signal is issued; therefore, in order to prevent stall and/or starvation phenomena in the simulation model, it is essential that Decode3 receives the abort message as soon as possible.

7.6.1.3 The Monitoring Path

To minimize the communication overhead imposed by the monitoring messages, the characteristics of the T-Rack may be exploited. As explained in section 7.5, the I/O process receives messages from both the Memory and the monitoring processes. Since the Tadpole transputer, where the I/O process resides,

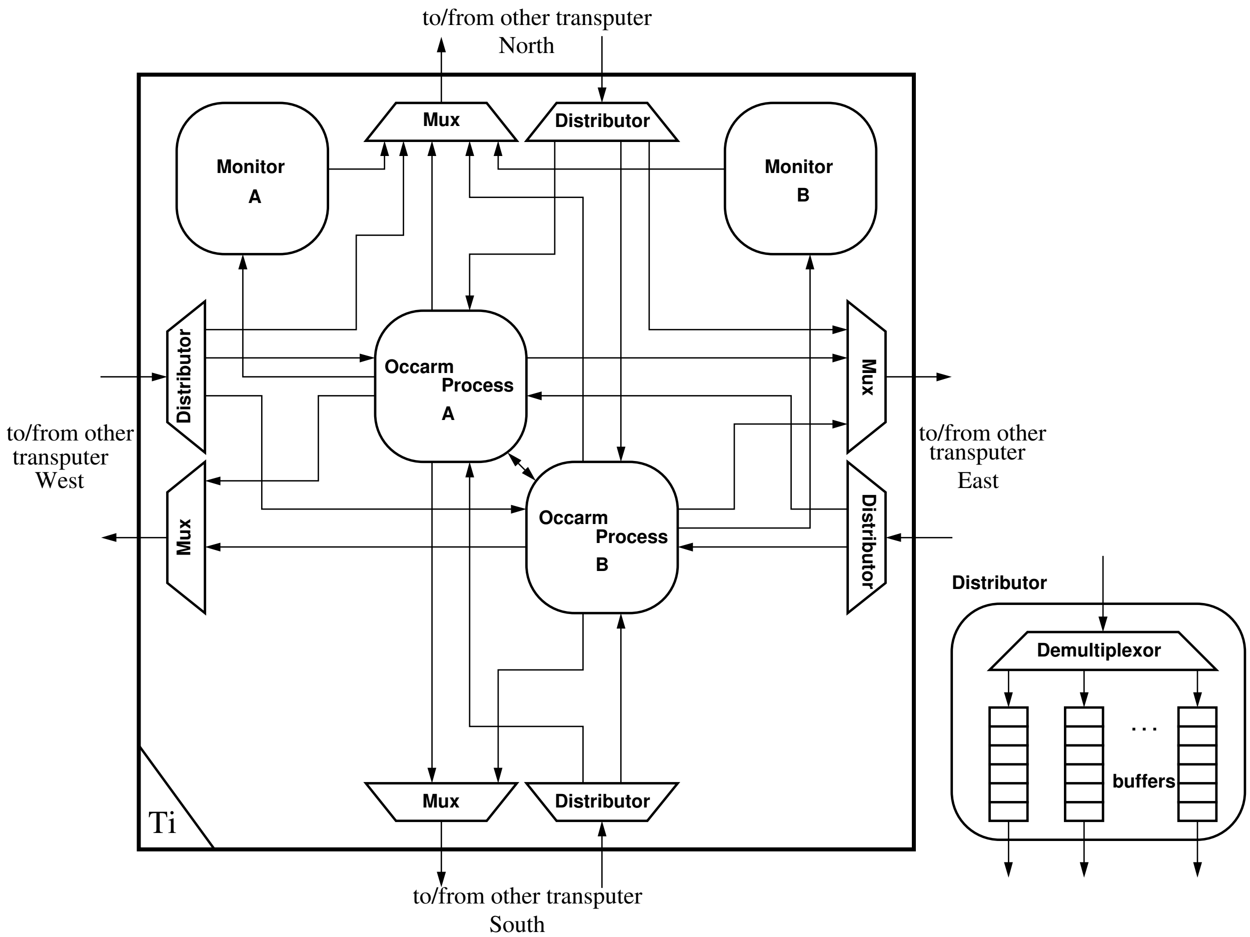


Figure 7.10: The Generic Simulator Node

is connected to both ends of the necklace, the interaction with the Memory process may take place via one end of the necklace, with the monitoring messages following the other direction towards the other end of the necklace. This scheme is depicted in figure 7.9, where the complete mapping of occarm onto the T-Rack is presented; transputers T0-T5 host the simulation model, while transputers T6-T63 are simply used to forward monitoring information.

7.6.1.4 The Generic Simulator Node

Figure 7.10 depicts a generic node of the distributed implementation of the simulator. Typically this will include a number of active processes together with the corresponding monitoring modules. Extra multiplexing/demultiplexing processes are included to allow the sharing of the transputer links; to prevent deadlock situations (which for example might occur if one of the transputer links is blocked by a message destined for a particular process, while this process is blocked waiting for a message that may follow the former on the link), extra buffering has been incorporated into the demultiplexing modules (i.e. the distributor process).

In practice, not all of the modules depicted in figure 7.10 will be included in a typical node.

7.7 Summary

This chapter has discussed issues related to the execution of the occarm model onto the T-Rack, a 64 transputer machine. Techniques to deal with the problems of monitoring, termination, mapping and load balancing have been presented; the techniques that have been presented regarding the monitoring of the occarm model are general and can be applied to any asynchronous architectural model which is based on the same philosophy as occarm.

The next chapter presents a set of results obtained from the execution of occarm on the T-Rack. These results are used for the validation of occarm.

Chapter 8

Validation of the Occarm Model

8.1 Introduction

The last two chapters have described the occarm simulation model and the associated environment which provides for the execution of occarm on the ParSiFal T-Rack. This chapter presents a set of quantitative results which provide the basis for the validation of occarm.

Simulation model verification and validation is a complicated[1] albeit important issue and is currently an active area of research. An overview of existing validation and verification approaches may be found in [Whit89], [Cars89], [Sado89], [Sarg92], [Sarg94], [Balc94] and [Balc94a].

The validation of occarm has been performed by comparing results produced by occarm with those produced by the Asim simulation model and concerns two different characteristics of occarm, namely :

- The accuracy of timing.
- The performance.

[1] Yucesan et al. [Yuce92] have shown that the verification of certain properties of discrete event simulation models is an NP-complete problem.

```
...
time();
for (Run_Index = 1; Run_Index <= Number_Of_Runs; ++Run_Index)
{

  Proc_5();
  Proc_4();

  Int_1_Loc = 2;
  Int_2_Loc = 3;
  strcpy (Str_2_Loc, "DHRYSTONE PROGRAM, 2'ND STRING");
  ...
} /* loop "for Run_Index" */
time();
...
Proc_4 ()
{
  Boolean Bool_Loc;

  Bool_Loc = Ch_1_Glob == 'A';
  Bool_Glob = Bool_Loc | Bool_Glob;
  Ch_2_Glob = 'B';
} /* Proc_4 */
...
Proc_5 ()
{
  Ch_1_Glob = 'A';
  Bool_Glob = false;
} /* Proc_5 */
...
```

Figure 8.1: A Section of the Dhrystone Synthetic Benchmark

8.2 Benchmark Programs

Within the AMULET project, for the verification and evaluation of the AMULET1 processor, the *ARM validation programs* [ARM] as well as the *Dhrystone* benchmark [Weic84] are used. The former are "toy" benchmarks which invoke different instruction types to test different parts of the design. Dhrystone is a synthetic benchmark which has traditionally[2] been used for the evaluation of computer architectures.

The main body of the Dhrystone program consists of a loop as depicted in figure 8.1. Using this benchmark, the performance of the simulated architecture

[2] A discussion on the different benchmarks used in connection with computer architecture research may be found in [Henn90] pp. 45-48.

Model	T_{before}			T_{after}		
	Value(ns)	Drift(ns)	Error(%)	Value(ns)	Drift(ns)	Error(%)
Occarm (Single)	39036	10313	20.9	114432	28834	20.13
Occarm (Multi)	39274	10075	20.42	116090	27176	18.97

Asim T_{before}= 49349ns
Asim T_{after} = 143266ns
Benchmark: Dhrystone (1 loop)

Table 8.1: Timestamp Drift

Model	Dhrystone Number	Error (%)
Occarm (Single)	13263.30	24.57
Occarm (Multi)	13018.30	22.26

Asim Dhrystone Number= 10647.69

Table 8.2: Dhrystone Numbers

is expressed in terms of the "Dhrystone number" which denotes the number of times that the loop is executed during a period of one second ("Dhrystones"). The "Dhrystone number" is calculated by sampling the current clock value before (T_{before}) and after (T_{after}) the execution of the loop and employing the formula:

$$Dhrystone.Number = \frac{10^9 * Number.Of.Runs}{T_{after} - T_{before}}$$

The clock values in the above formula refer to either real or simulated time depending on whether Dhrystone is executed on the physical processor or within a simulator respectively.

The functional correctness of occarm has been verified by (meta-)executing the complete set of the ARM validation programs.

For the validation of occarm, Dhrystone has been used for the following two reasons:

- Dhrystone, like all synthetic benchmarks, tries to match the average behaviour (i.e. the average frequency of operations and operands) of a large set of real programs; thus, the results obtained may be considered representative of the average behaviour of occarm too.

- Dhrystone is the benchmark that has typically been used for the evaluation of the AMULET1.

Within the ARM development environment used in the AMULET project, the "*time()*" function in the Dhrystone code (figure 8.1) is compiled as a write request to a particular address in memory; the memory responds by issuing a value which denotes the time that the write request is issued to memory. In occarm, this value is the timestamp of the request message to memory.

8.3 Accuracy

The accuracy of timing in occarm has been tested by comparing the results produced by occarm with those obtained from Asim (see chapter 6). These are the results that are used for the performance evaluation of AMULET1, namely the Dhrystone number, as well as the occupancy and stall periods of the AMULET1 pipelines (see section 7.3.1). Since the calculation of these values is based on the simulated time, they are particularly suitable to be used as a means for measuring the degree of timing accuracy of occarm.

Pipe (Figure)	Size	Occupancy (%)														
		1 Item			2 Items			3 Items			4 Items			5 Items		
		Single	Multi	Asim	Single	Multi	Asim	Single	Multi	Asim	Single	Multi	Asim	Single	Multi	Asim
IPipe(6.5)	5	25.35	21.66	14.78	35.53	33.04	26.07	27.13	30.71	37.37	11.00	14.35	19.71	0	0	0
PCPipe(6.4)	2	37.72	34.47	31.10	47.62	53.31	60.68									
XPipe(6.7)	3	37.13	36.88	40.30	9.24	9.43	10.59	0.55	0.92	1.20						
Dec1.RegB(6.8)	1	74.60	77.04	52.43												
Dec1.RegA(6.8)	1	7.99	7.85	5.04												
Dec2.OP(6.17)	1	89.62	90.44	69.06												
Dec2.Rsh(6.17)	1	0	0	0												
NGen(6.17)	2	10.05	10.36	3.53	3.95	3.87	0									
RB.OReg(6.13)	1	50.91	52.27	40.75												
RB.Wreg(6.13)	1	78.40	79.62	63.35												
RB.Ireg(6.13)	1	47.20	50.10	61.80												
RB.Afifo(6.13)	3	37.24	37.48	36.17	1.09	1.10	2.56	0	0	0						
RB.Mfifo(6.13)	4	22.63	22.45	22.90	3.04	3.27	2.91	0.69	0.65	0.77	0	0	0			
Dec3(6.8,6.18)	3	27.70	25.65	NA	52.26	52.36	NA	12.95	15.36	NA						
CPreg(6.18)	1	57.67	57.83	65.62												
OPreg(6.18)	1	56.21	56.42	55.41												
MemCP(6.5)	5	56.75	57.55	60.21	7.88	7.76	2.87	0	0	0	0	0	0	0	0	0
MRReg(6.5)	1	32.64	33.30	22.39												
MAReg(6.4)	1	57.29	57.49	42.17												
Dout(6.5)	3	10.31	10.48	13.09	0.43	0.44	0.09	0	0	0						
PCHLat(6.4)	2	95.65	95.67	90.11	0.60	0.77	0.76									
LSMreg(6.4)	1	6.79	6.70	4.42												
APipe(6.4)	2	5.17	5.30	7.65	0.17	0.16	0.20									
WReg(6.19)	1	14.31	15.23	18.71												
DataIn(6.5)	1	8.23	7.64	5.94												
RReg(6.18)	1	29.25	29.14	23.76												
ImmPipe(6.8)	2	40.17	41.32	27.63	13.15	14.30	4.13									

NA: Not Available

Table 8.3: AMULET1 Pipeline Occupancy (Dhrystone (1 loop))

Pipe (Figure)	Total Number of Messages			Stall Period (ns)								
				Average			Min			Max		
	Single	Multi	Asim	Single	Multi	Asim	Single	Multi	Asim	Single	Multi	Asim
IPipe(6.5)	992	1018	1079	12	12	12	12	12	12	12	12	12
PCPipe(6.4)	992	1018	1075	81.24	83.44	68.85	12	12	12	700	753	956
XPipe(6.7)	251	254	244	14.35	12.08	12.88	12	12	12	80	35	107
Dec1.RegB(6.8)	973	985	954	29.47	32.16	23.48	12	12	12	214	227	180
Dec1.RegA(6.8)	147	147	144	36.10	35.73	46.21	12	12	12	69	73	98
Dec2.OP(6.17)	973	985	953	58.35	61.12	67.33	12	12	12	322	325	409
Dec2.Rsh(6.17)	0	0	0	0	0	0	0	0	0	0	0	0
NGen(6.17)	117	117	113	12	12	12	12	12	12	12	12	12
RB.OReg(6.13)	973	985	953	43.51	44.16	25.53	12	12	12	134	197	182
RB.Wreg(6.13)	1034	1046	1012	12	12	12.70	12	12	12	12	12	15
RB.Ireg(6.13)	1034	1046	1012	16.03	17.03	50.89	12	12	12	170	178	393
RB.Afifo(6.13)	458	465	454	12	12	12	12	12	12	12	12	12
RB.Mfifo(6.13)	178	181	172	12	12	12	12	12	12	12	12	12
Dec3(6.18)	880	892	860	12.45	13.57	39.05	12	12	12	65	118	189
CPreg(6.18)	879	891	859	32.40	15.10	40.62	12	12	12	150	150	201
OPreg(6.18)	879	891	859	14.35	14.86	27.44	12	12	12	75	148	190
MemCP(6.5)	1156	1182	1258	12	12	12	12	12	12	12	12	12
MRReg(6.5)	1156	1182	1240	16.42	16.45	12	12	12	12	45	40	12
MAReg(6.4)	1343	1372	1421	29.56	29.03	42.51	12	12	21	128	114	92
Dout(6.5)	166	166	163	12	12	12	12	12	12	12	12	12
PCHLat(6.4)	991	1017	1079	12	12	12	12	12	12	12	12	12
LSMreg(6.4)	111	111	109	13.02	13.7	21.66	12	12	12	14	19	25
APipe(6.4)	73	73	73	12	12	12	12	12	12	12	12	12
WReg(6.19)	342	345	331	12	12	12	12	12	12	12	12	12
DataIn(6.5)	186	189	179	12	12	15.76	12	12	14	12	12	27
RReg(6.18)	716	725	700	12	12	12.53	12	12	12	12	12	26
ImmPipe(6.8)	499	507	482	12.26	12.53	12	12	12	12	29	35	12

Table 8.4: AMULET1 Pipeline Stalls (Dhrystone (1 loop))

In order to make the results obtained from the two different models comparable, care has been taken during the implementation of occarm to ensure that the propagation delays (in simulated time) incorporated in the occam processes, exactly match those of Asim. For modules described in Asim at the Register Transfer Level, this is straightforward; for Asim gate level descriptions[3], the equivalent occarm delays have been calculated by taking into account the propagation delays of all the circuit elements on the corresponding path[4]. Results from both, the single and multiple transputer configurations (referred to in the result tables as *Single* and *Multi* respectively) of occarm on the T-Rack have been obtained. These represent two different and arbitrary process schedulings, thus providing an indication of the effect that different process schedulings may have on the timing inaccuracy.

Table 8.1 illustrates the clock values T_{before} and T_{after} of the Dhrystone program as produced by occarm, while table 8.2 presents the Dhrystone numbers obtained from the aforementioned values; in these tables, the values "Drift" and "Error" refer to the comparison between the corresponding results from occarm and Asim.

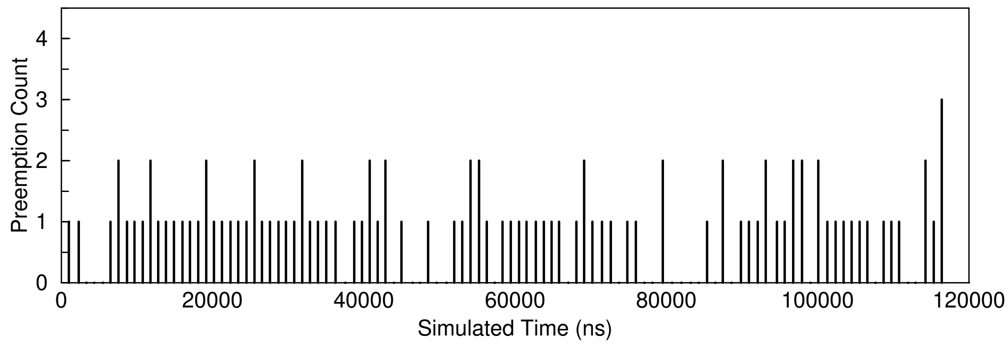
Tables 8.3 and 8.4 present the comparative results regarding the occupancy and the stall periods of the pipelines[5] of AMULET1 respectively.

As shown in table 8.2, the timing error with regard to Dhrystone values is 24.57% and 22.26% for the single and multiple transputer configurations of occarm respectively. These values may be considered reasonable and indeed acceptable at this level of simulation and at an early stage of the design process [WooJ94]; the same applies for the values obtained regarding the pipeline occupancy and stall periods, since, as it can be seen in tables 8.3 and 8.4, the

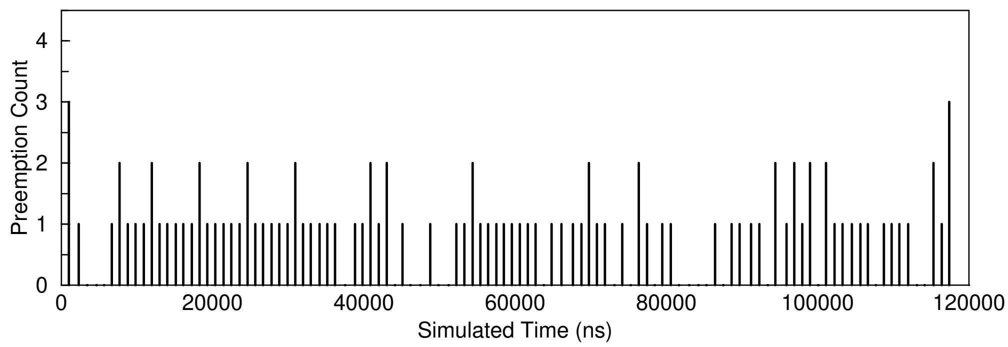
[3]This refers to modules that are described in terms of logic gates, latches and Event Control Blocks.

[4]These delays correspond to the CMOS implementation of AMULET1.

[5]A register is a pipeline of size one.



a) Single Transputer Implementation

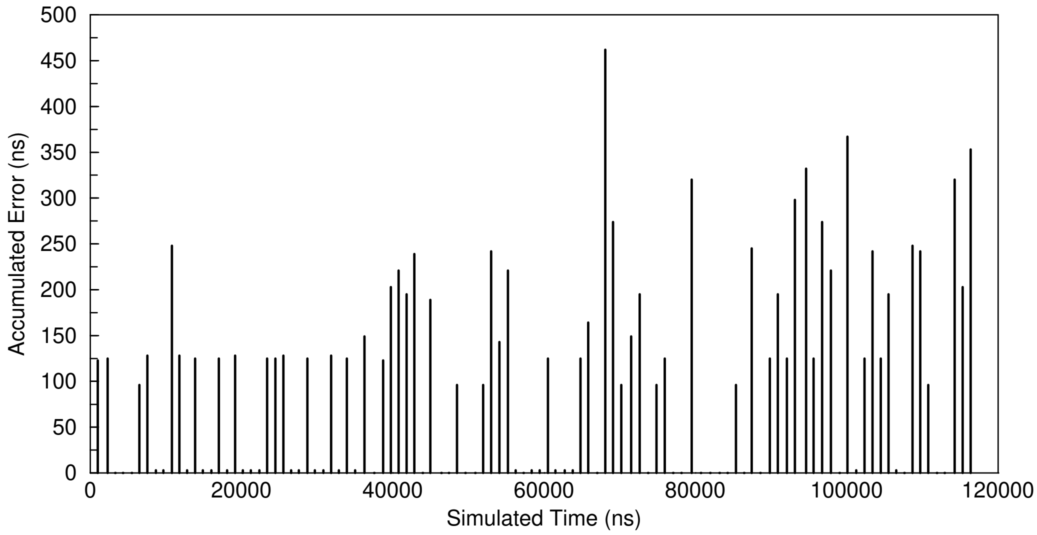


b) Multi Transputer Implementation

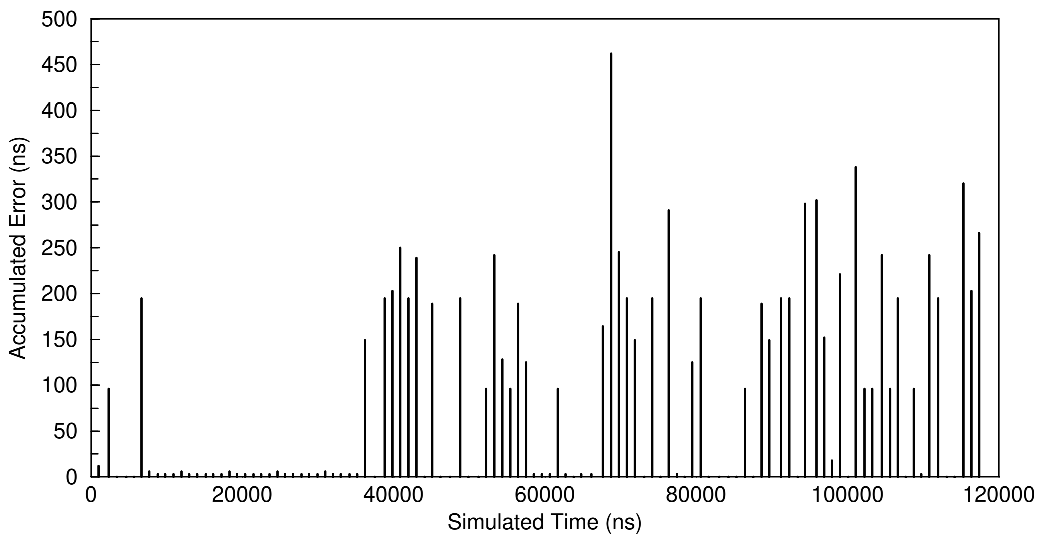
Figure 8.2: Decode1: Preemption Count (1 Dhrystone Loop)

disparities between the occarm and Asim results are insignificant.

It is important to note that the discrepancies between the results obtained from occarm and the Asim model are due, not only to the timing error introduced into the message timestamps by the occurrence of preemptions in occarm, but also, to the change in the system's behaviour that these preemptions cause. Preemptions in the arbiter processes change, not only the order, but also the type and number of events in the model. A preemption in the address interface (AddInt process) which involves a branch target address from the datapath (Wch channel, see section 6.9) affects the number of addresses issued to memory and consequently, the number of invalid instructions which will enter the system. Similarly, a preemption in Decode1 arbiter process means that a different number of invalid instructions will enter the datapath. This explains the disparities



a) Single Transputer Implementation



b) Multi Transputer Implementation

Figure 8.3: Decode1: Preemption Magnitude (1 Dhrystone Loop)

between the number of messages which enter each of the pipelines in the different models (see table 8.4).

In order to examine how preemptions are distributed over time, the number of times that a preemption is detected[6] (the preemption count) and the corresponding accumulated preemption magnitude (in *nanoseconds*) for each arbiter process

[6]A preemption in an arbiter process is detected each time a message is received with timestamp less than the timestamp of the last message on the other input link.

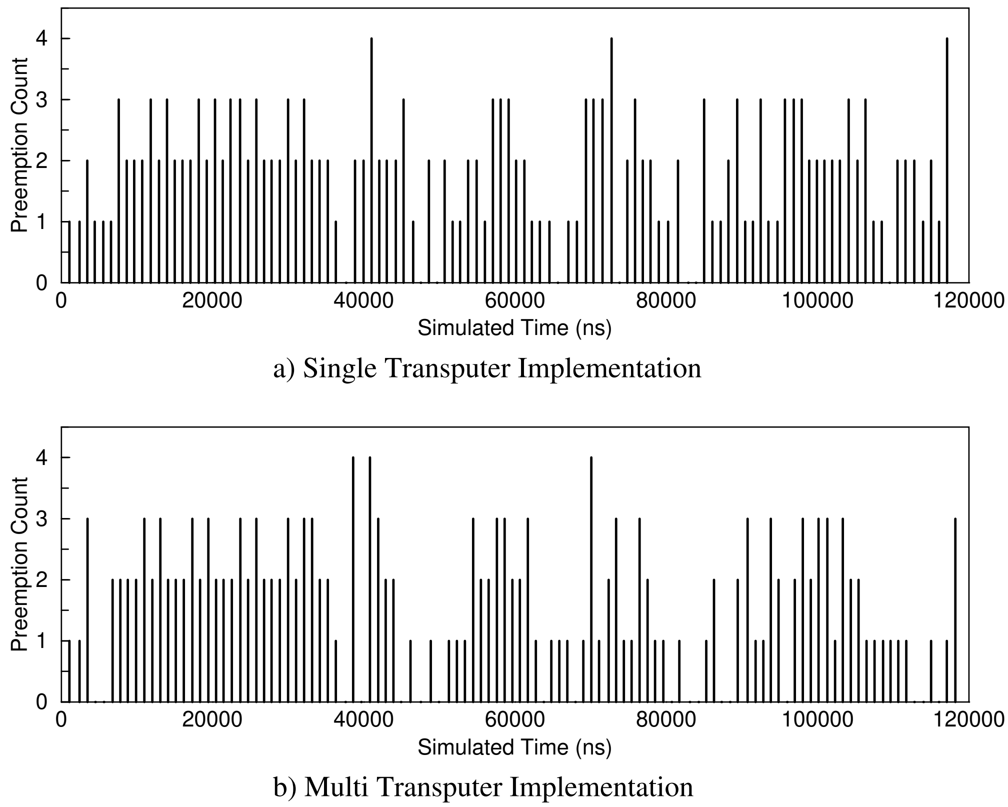
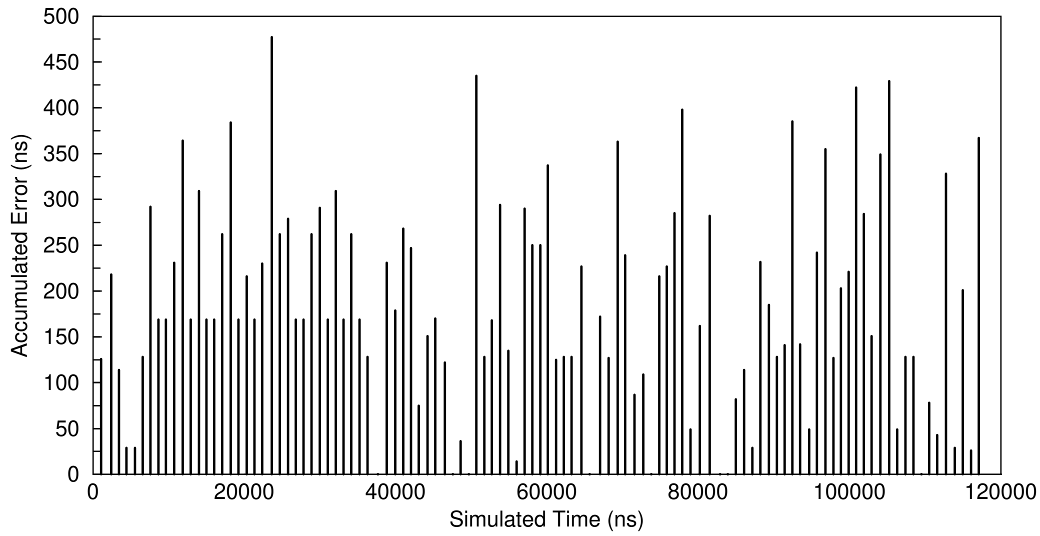


Figure 8.4: AddInt: Preemption Count (1 Dhrystone Loop)

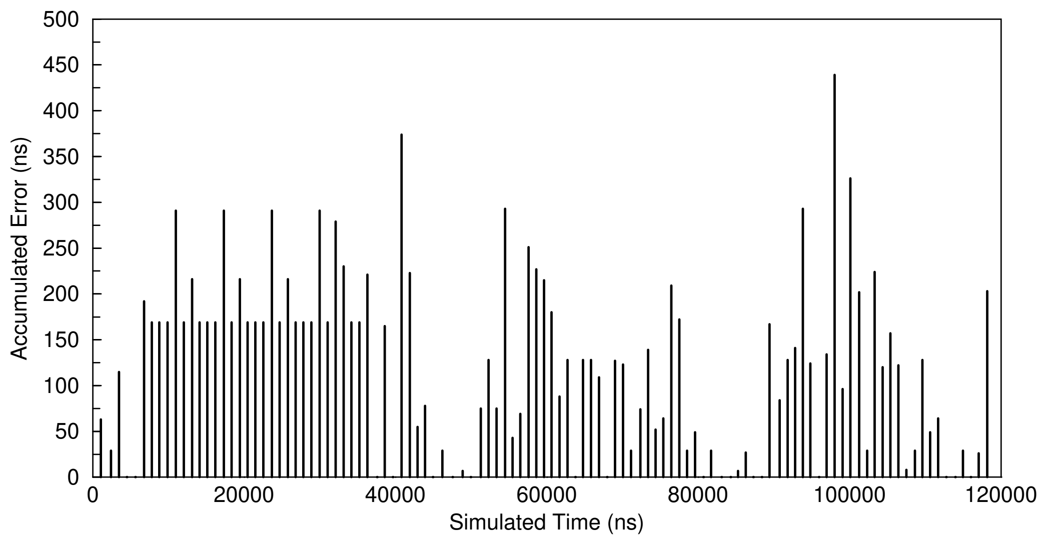
have been sampled at regular intervals[7]. Figures 8.2 - 8.7 present the obtained values in the form of impulse graphs. These figures indicate that preemptions take place at a low frequency, with the corresponding accumulated preemption magnitude being relatively small (ranging from less than 25ns to 475ns). Most of the preemptions occur in the AddInt process; this may be explained by the fact that the activity of the address interface (i.e number of messages arriving on the input channels of AddC process) is higher, thus increasing the probability of preemptions.

An interesting aspect with regard to preemptions which emerged from the

[7]In the current implementation of occarm, the sampling period is 10000ns (simulated time). However, the distributed and event-driven nature of occarm allows reports to be generated only after the report boundary instant has been crossed. in the impulse graphs, "accumulated error" refers to the total preemption magnitude within a sampling period.



a) Single Transputer Implementation



b) Multi Transputer Implementation

Figure 8.5: AddInt: Preemption Magnitude (1 Dhrystone Loop)

obtained results is that although locally, within each arbiter process, the accumulated timing error increases as the simulation progresses, the error which is incorporated in the timestamps of the messages in the model does not necessarily follow this pattern; certain individual messages may even have timestamps that exactly match the time that the corresponding events would occur in the absence of preemptions. This is illustrated in figure 8.8, where an example process graph

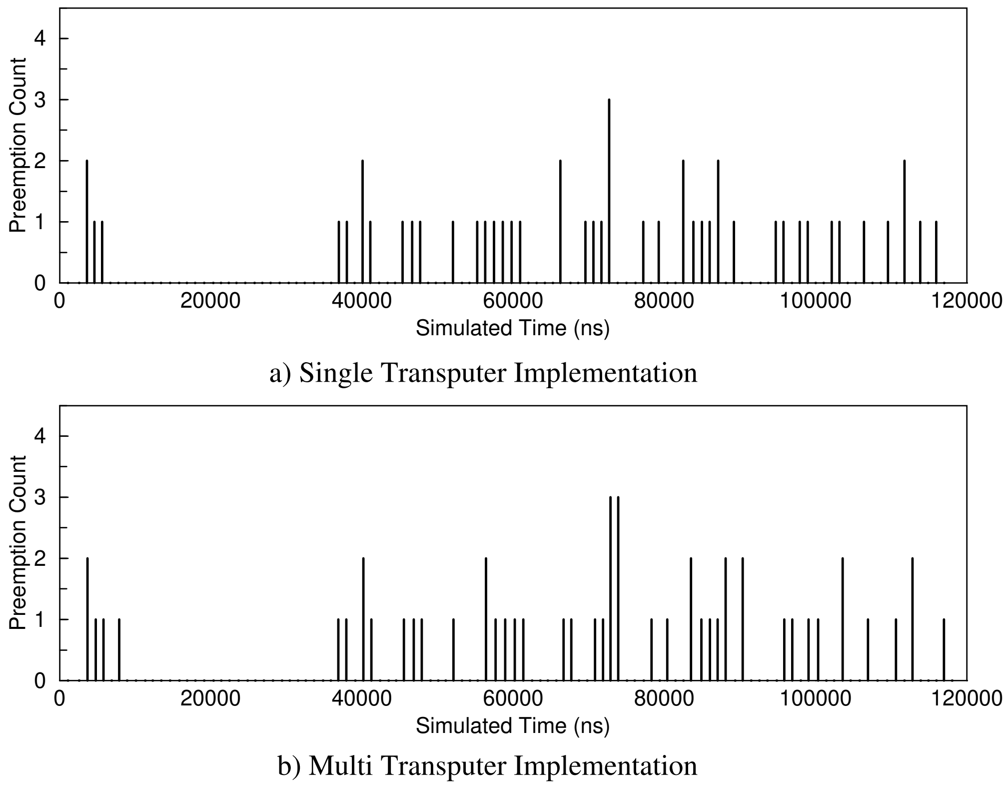
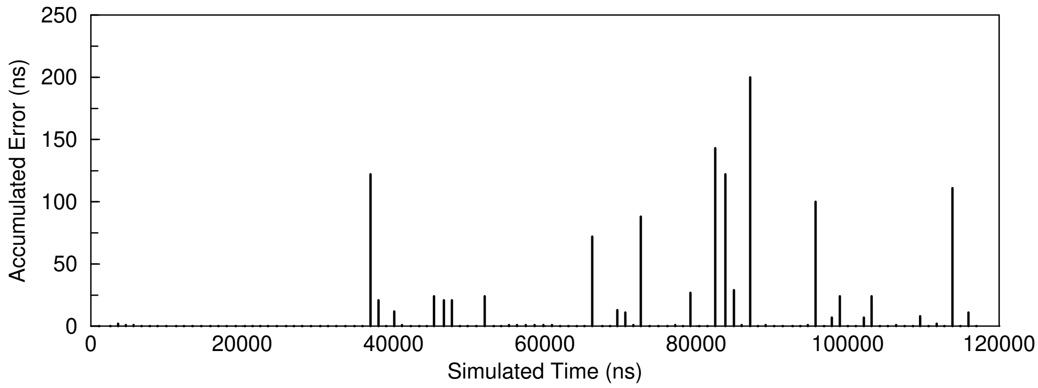


Figure 8.6: WrtCtrl: Preemption Count (1 Dhrystone Loop)

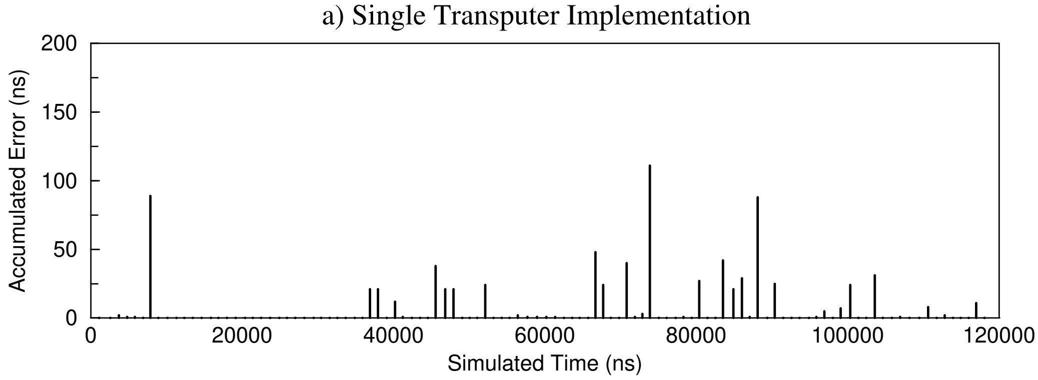
with two arbiter processes (*P1* and *P2*) is depicted. Assuming zero propagation delays within the processes, if no preemptions occur, messages would be sent to process *P3* in the order *a, b, c, d, e, f*, with timestamps 3, 5, 7, 10, 13 and 15 respectively. If preemptions occur, a possible sequence of messages to P3 could be *d, a, b, f, c, e*, with timestamps 10, 10, 10, 15, 15 and 15 and with timing error 0, 7, 5, 5, 8, 0 and 2 respectively.

8.4 Performance

With regard to performance, occarm and the Asim model are not directly comparable, since each describes the AMULET1 architecture at a different level, and the supporting machines (namely the T-Rack and any SPARC workstation) are intrinsically very different. Nevertheless, a comparative examination of the



a) Single Transputer Implementation



b) Multi Transputer Implementation

Figure 8.7: WrtCtrl: Preemption Magnitude (1 Dhrystone Loop)

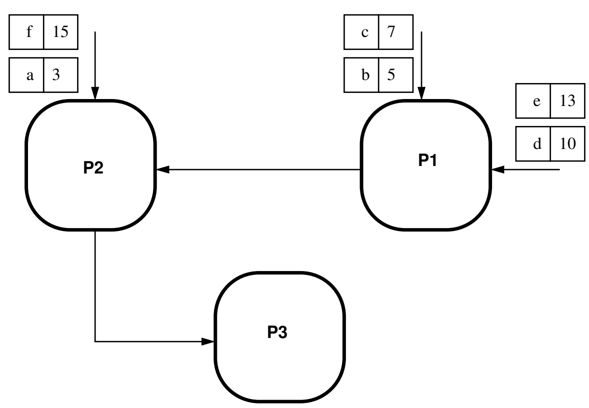


Figure 8.8: An Example Process Graph

performance of the two models may provide an indication of the impact that the use of the occam model may have to the duration (and cost) of the design cycle, within the boundaries of existing technology. Within the single transputer configuration (i.e. on a single 20MHz, T414 transputer), occarm requires on average[8] 1.72 minutes to execute one Dhrystone loop[9] when no monitoring traces are generated; as illustrated in table 8.5, this time is longer than that required by the Asim model executing on an IPX Sun workstation[10] by a factor of 1.16.

This is a reasonable and expected performance, for the execution of the occarm processes on a single transputer is performed not in a parallel but rather in a time sharing fashion and the large number of processes in the model[11] make the context switching overhead in the transputer significant.

Table 8.6 presents the performance of occarm for both, its single and multiple transputer configurations and for the different policies employed for the transportation of monitoring data (see section 7.3.1).

When no monitoring traces are generated, the distribution of occarm on to the seven transputers of the T-Rack yields a speedup of 1.69.

The "store and unload" transport policy allows a speedup of 2.26 to be achieved since, in this mode of operation, the performance of occarm on a single transputer drops by a factor of 2.45 compared to 1.83 in the multi transputer implementation. This difference in the performance drop may be attributed to the fact that the activation of the monitoring processes severely increases the frequency and, consequently, the overhead of context switching on the single transputer. The distribution of the monitoring processes onto multiple transputers alleviates this phenomenon as the context switching overhead is also distributed.

When the "immediate transport" policy is employed, the performance of both the single and multiple transputer configurations of occarm drops dramatically by

[8] The performance results presented in this section represent the average (i.e. the *mean* [Grah94], page 384) value derived from (10) runs of the simulators in question.

[9] This corresponds to the execution of 20 ARM machine instructions per second.

[10] The elapsed time for the Asim model has been obtained by taking into account the *user* and *sys* values provided by the Bourne shell "time" command.

[11] The current implementation of occarm consists of approximately 120 parallel processes.

Model	Elapsed Time (minutes)
Occarm(Single)	1.72
Asim	1.48

Benchmark: Dhrystone (1 loop)

Table 8.5: Asim versus Occarm (Single Transputer Implementation)

Transport Policy	Elapsed Time (minutes)		Speedup
	Occarm (single)	Occarm (multi)	
Tracing Off	1.72	1.02	1.69
Store and Unload	4.22	1.87	2.26
Immediate Transport	9.75	7.21	1.35

Benchmark: Dhrystone (1 loop)

Table 8.6: Performance of Occarm

5.67 and 7.07 respectively, allowing a speedup of only 1.35. This behaviour may be attributed to the operation of the I/O process. As explained in section 7.5, the I/O process acts as a multiplexor for messages arriving from both the Memory and the Monitoring processes. This introduces a major bottleneck in the system, which imposes the ultimate limit on the performance of the simulator. The large number of monitoring messages generated by the "immediate transport" policy occupy a large proportion of I/O process activity, thus reducing the rate at which instructions and data are supplied to the model; as a consequence, the processes of the model remain idle for substantial periods.

The speedups achieved by the distribution of occarm onto the multiple transputers of the T-Rack may be considered acceptable and reasonable [Birt94], but not satisfactory. The poor speedup achieved may be attributed to a number of factors related to the characteristics of both the simulated architecture and the machine that hosts the simulator.

- Amdahl's law [Amda67] specifies that the maximum possible speedup depends on the inherent parallelism of the executed system which may potentially be exploited[12]. In the case of AMULET1, the requirement for instruction compatibility with the synchronous ARM, has resulted in an asynchronous design with a very complex pipeline structure and, indicatively, limited parallelism. The performance of AMULET1 itself is 70% of the performance of of the synchronous ARM [Furb94]. The complexity of the AMULET1 architecture makes an analysis of the inherent parallelism of the design a complicated task which, as yet, has not been undertaken.

- Asynchronous architectures are communication bound systems and therefore the efficiency of the communication system is crucial. The complex irregular interconnection pattern of AMULET1's functional modules and the extra multiplexing/demultiplexing processes required to cope with the connectivity constrains of the Transputer and the T-Rack introduce major bottlenecks in the system which severely reduce the communication efficiency.

As explained in section 7.3.1, most of the time the simulation model will operate under the "store and unload" transport policy which permits the maximum speedup.

8.5 Summary

This chapter has presented a set of quantitative results regarding both, the accuracy and performance of the occarm simulation model. The accuracy of the model has been measured by comparing the results obtained from occarm with

[12]The estimation of the maximum possible speedup of a distributed simulation is currently an active area of research. Techniques which have been suggested for this purpose include the employment of a critical path analysis of a trace from a given simulation [Berr85] [Som89], and the treatment of the process graph as a queueing network model [Wagn89].

those provided by the Asim model; the results have been obtained by executing the Dhrystone benchmark on the two models. The distribution of preemtions over time and their effects on the messages' timestamps in the model have also been discussed.

The next chapter introduces an approach for eliminating occurance of preemptions within the framework of the proposed modelling approach.

Chapter 9

Addressing the Time Modelling Problem

9.1 Introduction

The results presented in the previous chapter confirm that the timing error introduced into the occarm simulation model by the lack of any synchronization between the parallel processes may be considered acceptable at this level of simulation and at the early stages of the simulation process.

Greater accuracy is needed, however, if the model is to serve as a tool for a more extensive and elaborate evaluation of the performance characteristics of the asynchronous architecture.

The requirement to test the architecture for potential deadlocks by modifying the delays in the system (see section 5.3.1.1) makes the accuracy requirement even more intense. Using the real time transputer clocks to change the relative scheduling of the occam processes, has the major drawback that small run time delays cannot guarantee the intended effects and behaviour in the model, as these delays are only approximate. Furthermore, run time delays have a direct effect on the performance of the simulator. Thus, large delays which would guarantee the planned process scheduling, would also affect the performance of the simulator; experiments with occarm have shown that the drop in performance can be significant.

This chapter describes a technique that has been developed to enforce time accuracy in occam models of asynchronous architectures.

9.2 Requirements

As discussed in chapter 5, the rationale behind using occam for modelling asynchronous architectures is the exploitation of the close relationship between the language semantics and the characteristics of the asynchronous system. Once the occam simulation model is constructed, any attempt to introduce time accuracy into it must preserve this modelling philosophy.

Furthermore, any complexity added to the model as a result of the synchronization protocol should be minimal. One of the purposes served by the occam model is to provide a description of the architecture's specification and operation; this information should not be obscured by extra functionality which is related only to the accurate operation of the model and not to the simulated architecture per se.

Any attempt to employ one of the existing synchronization protocols surveyed in chapter 3, would force it to depart from the modelling philosophy which provided its original basis. Therefore, to meet the aforementioned requirements, a novel synchronization protocol has been devised, namely the *Program Driven Synchronization Protocol* (*PDSP*). This is a conservative protocol which is based on a combination of the exploitation of the characteristics of the simulated system and the employment of Null messages to achieve deadlock avoidance, while maintaining the philosophy of the model intact. It seeks to enable the development of accurate arbiter models involving only the processes required for this purpose. The processes of the model remain entirely data driven.

9.3 The Program Driven Synchronization Protocol (PDSP)

9.3.1 The Basis

Von Neumann computer architectures, synchronous or asynchronous, are deterministic systems; they accept as input instructions which they execute sequentially in a specific and predefined order. Each instruction defines the steps that are required for its execution as well as the behaviour of each functional module of the architecture. Consequently, the kind and sequence of events that occur in the system are determined at any time by the executing instructions.

This ability to predict events in the architecture, based on the information provided by the program under execution, forms the basis of the Program Driven Synchronization Protocol; by examining the instructions being executed, the arbiter processes of the simulation model can determine whether an event is expected on a particular input link and thus whether their blocking upon this link would result in a deadlock.

The key concept in the Program Driven philosophy is the "Instruction Lookahead Set" which is defined as follows:

Definition 1 *The* **Instruction Lookahead Set** *(***ILS***)of a link λ is the set of instructions*[1] *whose execution will potentially result in an event occurring on λ:* $\mathbf{ILS}_\lambda = \{$*Instruction* $\mathbf{I}$*:* $\mathbf{I}$ *generates an event on link* $\lambda\}$.

The Instruction Lookahead Set of any particular link in the system is directly

[1] An instruction **I** is referred to as an ILS_λ *instruction* if and only if $I \in ILS_\lambda$.

defined by the architecture's specification and thus, may become available to the arbiter processes of the simulation model in advance. Based on the ILS of their input links, arbiter processes may directly make decisions regarding the potential arrival of messages, provided of course that they are also informed of the instructions being executed in the system.

9.3.2 The Rules

Based on the Instruction Lookahead Set defined above, the behaviour of arbiter processes with regard to message consumption may be specified as follows:

Rule 1 *An arbiter process* Π *is allowed to block and wait for an event on its input link* λ *during the execution of an instruction* $\mathbf{I}$ *if and only if* $\mathbf{I} \in \mathbf{ILS}_\lambda$.

The above rule ensures that arbiter processes block only for instructions that are likely to generate the corresponding events. However, depending on the status of the system, during the instruction's execution such an event might not occur; in this case Null messages are required otherwise the arbiter process will become blocked and the simulation model will deadlock. The following rule is concerned with the production of Null messages:

Rule 2 *A Null message will be sent to link* λ *of the arbiter process* Π *if and only if* Π *expects an event on* λ *based on the* $\mathbf{ILS}_\lambda$*, and for the current state of the system the event will not be generated.*

The two rules above, specify the behaviour of arbiter processes and their peers, when their interaction depends on the executed instructions. However, not all events in an asynchronous system occur in an instruction dependent fashion. Indeed, certain parts of the system may operate autonomously, irrespective of which instructions are being executed; the PC loop in the AMULET1 processor is an example of such an autonomous unit. In this case it is the state of the simulated system that dictates the behaviour of the arbiter process:

Rule 3 *An arbiter process* Π *is allowed to block and wait for an event on its input link* λ *which fires in an instruction independent way, if and only if the state of the system guarantees that a message will be issued on* λ.

9.3.3 The PDSP Arbiter Process

The basic functionality of an arbiter process with regard to the Program Driven Synchronization Protocol is depicted in figure 9.1.

Upon receiving a message on one of its links (e.g. *msg1* message on *In1*) the arbiter invokes the *Select* process to determine whether the processing of this message would cause a preemption.

If there is a pending message *msg2*, already received from the other input, then the message with the minimum timestamp is selected to be processed and forwarded to the arbiter process's output; if both timestamps have the same value, the selection is made in a random fashion to emulate the behaviour of the corresponding hardware arbiter.

If however, no pending message exists, but a positive prediction (based on the Instruction Lookahead) is made regarding its potential arrival, the arbiter process blocks and waits until this second message arrives. The arrival of this message provides the arbiter process with the information required to proceed with its operation and enable *Select* to make a decision, namely the next timestamp on its other input link.

```
PROC PDSP_Arbiter()
  PROC Select(msg1, msg2)
    SEQ
      IF
        msg2_pending=TRUE
          SEQ
            IF
              timestamp(msg1)<timestamp(msg2)
                SEQ
                  process(msg1)
                  msg1_pending:=FALSE
              timestamp(msg1)>timestamp(msg2)
                SEQ
                  process(msg2)
                  msg2_pending:=FALSE
              timestamp(msg1)=timestamp(msg2)
                SEQ
                  make_random_selection(msg1,msg2)
        msg2_pending=FALSE
          SEQ
            IF
              msg2_expected=TRUE
                SEQ
                  In2?msg2
                  msg2_pending:=TRUE
              msg2_expected=FALSE
                SEQ
                  process(msg1)
                  msg1_pending:=FALSE
  :
  WHILE TRUE
    SEQ
      IF
        msg1_pending=TRUE
          SEQ
            Select(msg1,msg2)
        msg2_pending=TRUE
          SEQ
            Select(msg2,msg1)
        TRUE
          SEQ
            ALT
              In1?msg1
                msg1_pending:=TRUE
              In2?msg2
                msg2_pending:=TRUE
:
```

Figure 9.1: The PDSP Arbiter Process

9.3.3.1 Improving PDSP Performance

The basic algorithm described in the previous section (figure 9.1) enables arbiter processes to receive and process messages arriving on their input links in increasing timestamp order, always selecting the message with the smallest timestamp;

this guarantees the accurate, preemption-free operation of the simulation model.

However, this process does not ensure that the concurrency of the simulated system is sufficiently exploited to increase the potential of the model for high performance. Indeed, as soon as it predicts that a message is expected on one of its input links (e.g. In2), the arbiter process will stop accepting any messages arriving on its other input link In1 until the expected message on In2 arrives. As a consequence, all the processes which are part of the path that leads to In1 will block and wait, and the pipelines at the output side of the arbiter process will starve; during this time, large parts of the simulator will remain idle.

A solution to this problem is to provide arbiter processes with some indication as to when, in the simulated future, an expected message will actually arrive. This information will enable them to consume a number of events occurring on In1 link before they block on In2 increasing thus the concurrency of the simulation model. This information can be obtained by taking into account the propagation delays in the architecture being simulated. An event generated by an instruction will propagate through a number of pipeline stages before it reaches an arbiter. The path followed by the event is completely defined by its parent instruction; the latency of the path however at any particular time, depends on the number of elements in the micropipelines involved and thus, it is non-deterministic.

Consequently, it is not feasible to know in advance the exact time required for an event to propagate through a given micropipeline. However, there is a lower bound to this time, namely the latency of the micropipeline when, at the moment of the event's entry, it is empty. Based on this observation, the Minimum Latency Lookahead, may be defined:

Definition 2 *The* **Minimum Latency Lookahead** *of a link* λ $\mathbf{MLL}_\lambda$, *is defined as the total propagation delay of the path leading to* λ*, when the pipelines of the path are empty:*

$$\mathbf{MLL}_{\lambda} = \sum_i d_i$$

d_i: Propagation delay of the i-th pipeline stage in the path.

The Instruction Lookahead Set of a link informs the corresponding arbiter process *whether* a message should be expected on that link; the Minimum Latency Lookahead reveals *when* in the simulated future the expected message may arrive. Based on the Minimum Latency Lookahead, the following rule may be specified:

Rule 4 *An arbiter process Π will not process a message μ_1 received on its input link λ_1 but instead it will block and wait for a message μ_2 expected on its other input link λ_2, if and only if the timestamp of μ_2 as predicted by the $MLL_{\boldsymbol{\lambda_2}}$ is less than or equal to the timestamp of μ_1.*

Rule 4, combined with the ALT statement in the main loop of the PDSP arbiter process, will enable arbiter processes to process messages with appropriate timestamps as soon as they arrive.

Figure 9.2 depicts the functionality of the arbiter process when the MLL is taken into account.

9.3.4 The Limitations

The Program Driven Approach is based on the exploitation of the Instruction Lookahead properties of the simulated architecture. Such an exploitation presupposes that arbiter processes have knowledge as to which instructions are being executed. If this knowledge is not directly available, then an appropriate mechanism needs to be devised to provide arbiter processes with this information. Generally the functionality of the architecture being modelled will make the development of such a mechanism feasible. Otherwise the instruction lookahead properties of the system can not be exploited and the PDSP rules can not apply.

```
PROC PDSP_Arbiter()
  PROC Select(msg1, msg2)
    SEQ
      IF
        msg2_pending=TRUE
          SEQ
            IF
              timestamp(msg1)<timestamp(msg2)
                SEQ
                  process(msg1)
                  msg1_pending:=FALSE
              timestamp(msg1)>timestamp(msg2)
                SEQ
                  process(msg2)
                  msg2_pending:=FALSE
              timestamp(msg1)=timestamp(msg2)
                SEQ
                  make_random_selection(msg1,msg2)
        msg2_pending=FALSE
          SEQ
            IF
              msg2_expected=TRUE
                SEQ
                  IF
                    timestamp(msg1)< MLL_timestamp(msg2)
                      SEQ
                        process(msg1)
                        msg1_pending:=FALSE
                    timestamp(msg1) >= MLL_timestamp(msg2)
                      SEQ
                        In2?msg2
                        msg2_pending:=TRUE
              msg2_expected=FALSE
                SEQ
                  process(msg1)
                  msg1_pending:=FALSE
  :
  WHILE TRUE
    SEQ
      IF
        msg1_pending=TRUE
          SEQ
            Select(msg1,msg2)
        msg2_pending=TRUE
          SEQ
            Select(msg2,msg1)
        TRUE
          SEQ
            ALT
              In1?msg1
                msg1_pending:=TRUE
              In2?msg2
                msg2_pending:=TRUE
:
```

Figure 9.2: PDSP: Taking MLL into Account

9.4 Applying PDSP to Occarm

In order to demonstrate its applicability, the Program Driven Synchronization Protocol was employed to develop models of the arbiter processes in occarm which would provide accurate modelling of time.

As already mentioned, occarm makes use of three arbiter processes, namely AddC, Dec1CtrlA and WrtCtrl, which form part of the address interface, primary decode and write bus control units of the AMULET1 respectively.

9.5 The Address Interface Arbiter

As described in section 6.3.2, the AddC process employs arbitration to allow addresses arriving from the datapath on the Wch channel to break the PC loop and gain access to the MAReg. The Instruction Lookahead Set of the Wch channel is[2]:

$\mathbf{ILS}_{Wch}$={*B, BL, SWI, LDR, STR, LDM, STM, Data Processing with PC as Dest. Reg.*} .

The other input of the arbiter, namely the PCch, forms part of the PC loop and is fed with PC addresses from the PC Pipe. The operation of the PC loop is autonomous and independent of the operation of the rest of the processor. Thus, on the PCch channel there will be a continuous, instruction independent flow of messages.

9.5.1 Providing Instruction Lookahead Information

AddC is an example of an arbiter process which has no direct knowledge regarding executing instructions. An address produced by AddC is sent to memory following the path from AddC, through MAReg and DataInt (MemCtrl) to memory. If it is

[2]See appendix A.

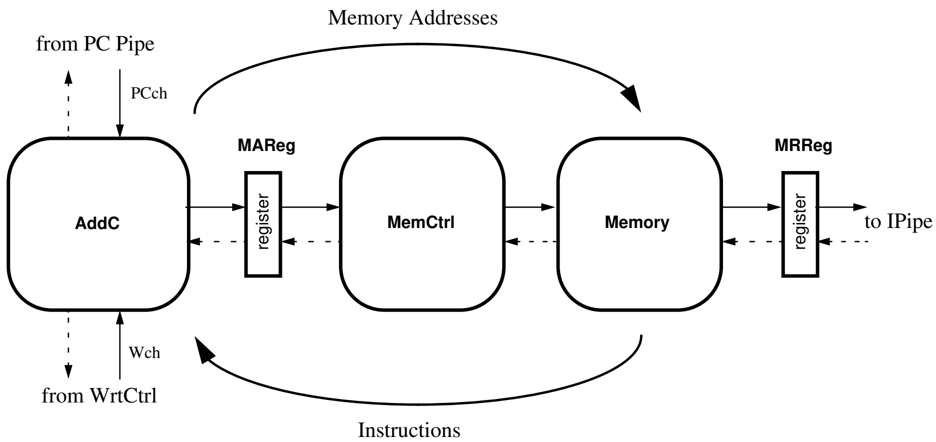


Figure 9.3: Providing Instruction Lookahead Knowledge to AddC

an instruction address, the instruction message from memory enters the processor following the path from memory through DataInt (IPipe) to Decode1.

AddC is not in the path followed by the instruction and therefore has no direct information as to which instructions have entered the system; in order to apply PDSP a mechanism is required to provide AddC with this information.

A neat and efficient solution is to take advantage of the hidden links in the above paths, namely the contra flow of the acknowledgement messages. Acknowledgement messages are sent from register to register through control processes in the model. In the first path above, an address message produced by AddC will propagate to memory and from there to MRReg; MRReg will generate an acknowledgement message which will follow the flow back to AddC (figure 9.3). This acknowledgement message can be used to carry the corresponding instruction to AddC; no communication overhead is generated as the acknowledgement messages would be sent anyway.

Clearly, after issuing an instruction address to memory, AddC will receive and process a number of PC messages arriving on the PCch link from the PC Pipe before it receives the acknowledgement with the corresponding instruction.

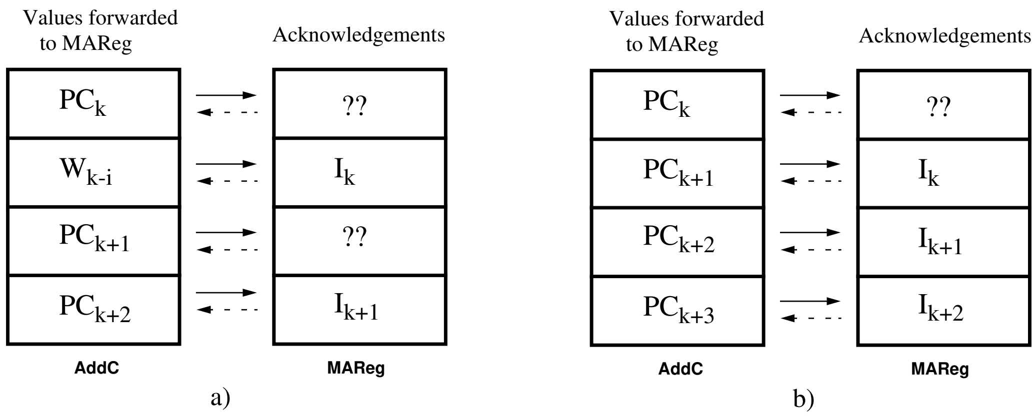


Figure 9.4: The Arrival of Instruction Lookahead Information

However, during this period no preemption will occur. Indeed, the acknowledgement message carrying an ILS_{Wch} instruction from memory will arrive at the AddC, just before the first or the second PC value following this instruction's address on the PCch channel is received by the AddC process. This is illustrated in figure 9.4. The acknowledgment carrying an instruction I_k will be issued by MAReg in response to its receiving, from AddC, the next memory address message immediately after the instruction's address PC_k. If, after PC_k, AddC selects and forwards a message from Wch channel (namely message W_{k-1}), then I_k will reach AddC immediately before this process is ready to receive address PC_{k+1} (figure 9.4a); otherwise I_k will arrive after PC_{k+1} has been forwarded to MAReg and before PC_{k+2} is processed (figure 9.4b). The value of PC_{k+2} is $PC_k + 8$. Therefore, it is the R15 value which, according to the PC+8 rule, will rendezvous with I_k in Decode1 before they both enter the datapath (see section 6.5). Since the corresponding message W_k on the Wch channel will be generated as a result of the execution of I_k, its timestamp t_{W_k} will be equal to: $max(t_{PC_{k+2}}, t_{I_k}) + d_{datapath}$, where t_{I_k} is the timestamp of the instruction when it reaches Decode1 and $d_{datapath}$ is the total propagation delay within of the datapath. Therefore, $t_{PC_{k+2}} < t_{W_k}$.

With the AddC process informed of the instructions entering the system, the application of of PDSP with regard to the prefetching loop arbiter becomes feasible. Indeed this information may be used by AddC in order to make decisions regarding the potential arrival of messages on its input links.

9.5.2 The PCch Link

The PCch channel carries the acknowledgement signal from the PC Pipe. As already mentioned, this signal is issued in a continuous, instruction independent fashion, each time the current PC circulating in the PC loop is latched by the first register of the PC Pipe (the PC0 register, see section 6.3.2). The issuing of the acknowledgement signal closes the prefetching loop, thus triggering a new cycle. Therefore, a message on the PCch channel will be expected by AddC, if a new PC value has been forwarded to be incremented and sent to the PC Pipe.

However, if the PC Pipe is full when the request signal to the PC0 register is issued, the circulating PC value will not enter the PC Pipe until at least one element from the PC Pipe is removed; this will empty PC0 register, enabling it to latch the pending PC value and consequently to issue the acknowledgement signal on the PCch channel.

The PC Pipe becomes full as a result of the datapath being stalled. The role of the PC Pipe is to provide the processor's datapath with the R15 values required for the execution of instructions (see section 6.3.1.2); its output is connected to the first stage of the datapath, namely the primary decode. If, for any reason, the datapath stalls, instructions will start to backlog filling the datapath up to the primary decode; consequently, further instructions with their associated PC values will be prevented from entering the datapath. As a result, the PC Pipe will become full and will remain so for as long as the datapath is stalled. During this period no further PC values will be allowed to enter the PC Pipe and thus no acknowledgment signal will be issued on the PCch channel.

This behaviour of the system may lead to deadlock situations in the simulation model when the PDSP algorithm is employed. This is due to the fact that not only the input, but also the output links of the datapath are connected to the address interface, thus forming closed communication paths as illustrated in figure 9.5. If, therefore, the datapath stalls as a result of its waiting to interact with the AddC process, and this process is blocked waiting for an acknowledgement message from the PC Pipe which however will never arrive because the PC Pipe has become full as a result of the datapath's stall, a deadlock will occur.

This situation may arise when, as a result of the execution of the ILS_{Wch} instruction, a message with timestamp t_W is produced by the datapath on Wch link. Obeying the PDSP algorithm, AddC receives this message but does not process it immediately. Instead, it turns its attention to serving the PCch channel, continuously reading acknowledgement messages arriving from the PC Pipe; this operation will continue for as long as the timestamp t_{PC} of the message on the PCch is less that t_W.

Each acknowledgement message received on the PCch channel carries the PC value which has just entered the PC Pipe. This value is forwarded by the AddC to subsequent stages of the PC loop, to be incremented and eventually channelled through the PC Pipe to the datapath. If, during this operation of the AddC process, the datapath stalls, then the PC Pipe will eventually fill up and, as a result, will be unable to issue any more acknowledgement messages to AddC. If, before the PC Pipe fills up, an acknowledgement message with timestamp t_{PC} *greater* than t_W is issued on the PCch channel, the operation of the simulator will proceed smoothly and without any implications. AddC will process the pending message from the Wch link, since it has the smallest timestamp; this will unblock the datapath enabling thus more instructions to

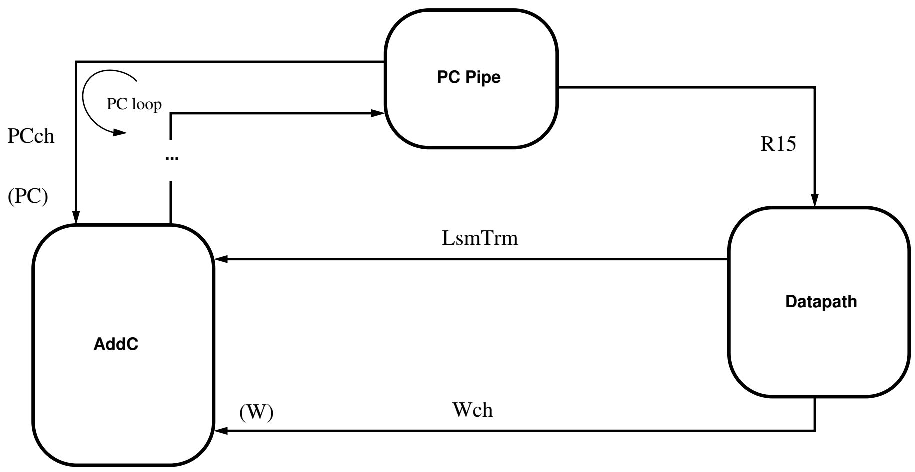


Figure 9.5: The Address Interface - Datapath Loop

enter it and, consequently, allowing the PC Pipe to output its contents.

If however *no* message with timestamp greater than t_W is received from the PCch channel before the PC Pipe fills up, the simulator will deadlock: AddC will remain blocked waiting for the next acknowledgment which will never arrive since the datapath will remain blocked and the PC Pipe full.

It is essential therefore that the AddC process be able to decide whether it should wait for yet another message from the PCch or whether the PC Pipe has become full and thus no more messages will be sent on the PCch link.

During the execution of an ILS_{Wch} instruction, the datapath may stall in the following cases:

- If the datapath fills up; this will occur as a result of Ctrl3 and WrtCtrl processes waiting for the abort/no abort and Wlx signals respectively.
- If the ILS_{Wch} instruction is followed by register read operations which refer to locked registers.
- If the ILS_{Wch} instruction is followed by instructions which activate the ALUgo signal.

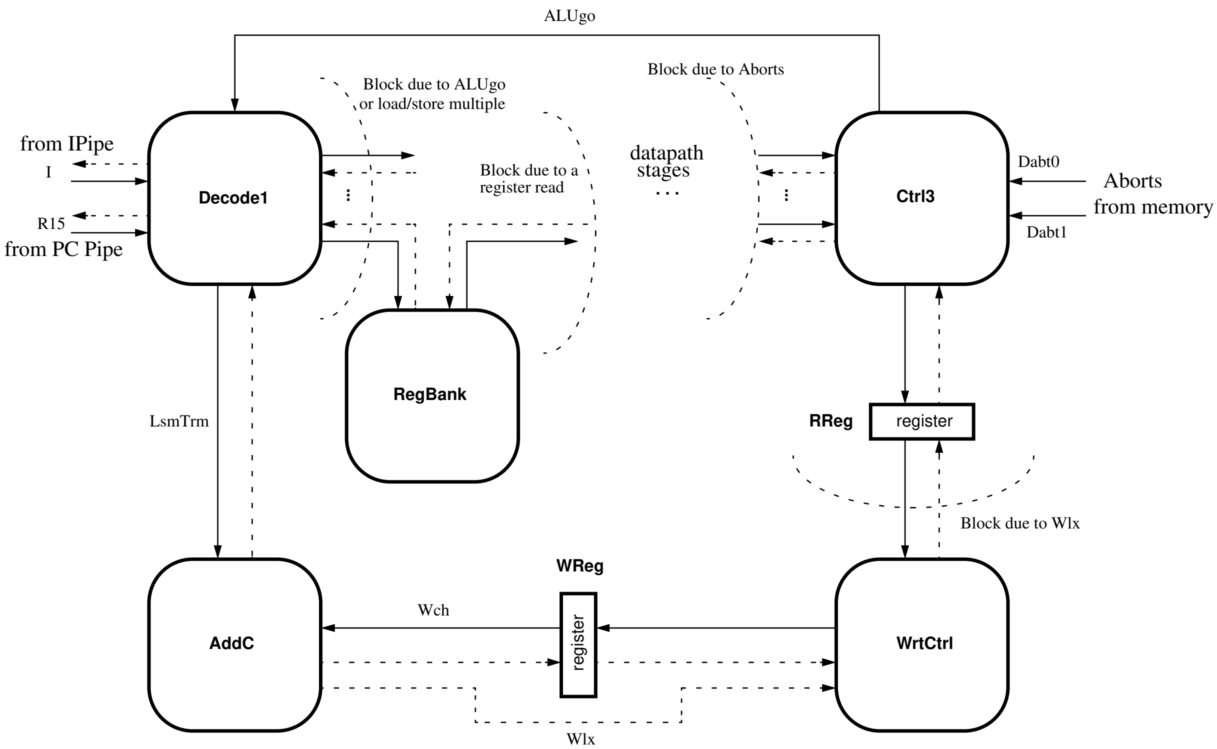


Figure 9.6: Stalling of the Datapath

- During the execution of load/store multiple instructions.

Figure 9.6 illustrates the blocking of the datapath, as a result of the aforementioned cases.

9.5.2.1 Filling of the Datapath

Address values produced by the datapath as a result of the execution of ILS_{Wch} instructions are forwarded to the AddC process through the WrtCtrl process (see figure 9.6). As explained in section 6.9, after forwarding the address message to AddC (via WReg register), WrtCtrl will block until both the acknowledgement from the Wreg and the Wlx signal from AddC are issued, whereupon WrtCtrl will continue its operation producing an acknowledgement message to the RReg register.

The acknowledgment from the WReg register will be issued as soon as it latches the address message, in parallel with the propagation of this message to AddC via the Wch channel. The Wlx channel however will not fire until AddC has processed the address message and has forwarded it to the Memory Address Register (see section 6.3.2); during this period, WrtCtrl will remain suspended, thus blocking the exit of the datapath.

For ILS_{Wch} instructions which activate the abort signals from memory, WrtCtrl will be suspended in a similar fashion, the datapath however will block before that point, due to Ctrl3's waiting to receive those signals from memory (figure 9.6).

The execution of instructions generates a flow of messages towards the exit of the datapath. Thus if either WrtCtrl or Ctrl3 are blocked, the datapath will gradually fill up and eventually stall.

The execution of a single-cycle instruction in a micropipelined architecture consists of the decoded instruction word being propagated to consecutive stages of the datapath, with each stage performing a different, predefined operation to produce an intermediate result. Thus, at any time a single-cycle instruction will occupy only one stage of the datapath. For multi-cycle instructions, a separate message will be generated and forwarded down the datapath for each cycle.

Hence, as a general rule, the datapath pipeline will become full if the total number of cycles of the instructions which have entered the datapath after an ILS_{Wch} and are being executed is equal to the number of stages left in the datapath. Table 9.1 presents the number of stages in the datapath for different ILS_{Wch} instructions.

As described in section 9.5.1, the AddC process is informed of the instructions which have entered the processor at any particular moment. However, not all instructions which enter the datapath will eventually be executed; instructions with colours not matching the operating colour of the processor or instructions

ILS_{Wch} Instruction	No. of Stages
B	N + 1
BL	N + 1
SWI	N + 1
LDR	N
STR	N
LDM	N
STM	N
Data. Proc. R15	N + 1

For the current implementation of AMULET1, N=3.

Table 9.1: PDSP: Number of Free Stages in the Datapath

which fail their condition codes will be discarded as invalid (see section 6.5). An instruction which is discarded will free the registers of the datapath which it would otherwise occupy, thus allowing more instructions to enter the datapath. Therefore, in order to enable the AddC process to determine whether the datapath has become full and as a result has stalled, it is necessary to provide this process with information regarding which of the instructions which enter the datapath are executed; AddC is aware of both the colour and the condition field of instructions entering the datapath as these form part of the instruction word which is provided to AddC.

Condition Codes Failure

In AMULET1, the test of the condition flags is performed by the CPSR unit which, in the occarm model, has been incorporated in the functionality of the Ctrl3 process (see section 6.8.1).

As already mentioned, an ILS_{Wch} instruction which has the potential to cause a memory abort will keep Ctrl3 process blocked until AddC lets the address message from Wch channel through to memory (figure 9.6). As a result, during this period, none of the instructions following the ILS_{Wch} instruction in the datapath will be able to enter Ctrl3, and possibly be rejected due to their failing their condition codes. Therefore, in order to determine whether the datapath is full, AddC must take into account *all* instructions which enter the datapath irrespective of their colour.

For ILS_{Wch} instructions which do not activate the abort signal however, Ctrl3 does not block waiting for memory's response. As soon as the memory address produced by Ctrl3 as a result of the execution of the ILS_{Wch} instruction is forwarded to the RReg register process (i.e. as soon as the RReg has issued the corresponding acknowledgment message), Ctrl3 is free to receive subsequent instructions and, possibly, reject them.

The operation of Ctrl3 will proceed until an instruction is encountered whose execution produces a result to be forwarded to RReg. Since RReg is blocked waiting for the acknowledgment from AddC (through WrtCtrl), the production of the result will block Ctrl3 as well. Henceforth, no further instructions will enter Ctrl3 until AddC processes the pending address message and issues an acknowledgment.

The *Current Processor Status* (CPS) at the time of the execution of the current ILS_{Wch} instruction may be sent to AddC with the address message; knowledge of the CPS will enable AddC to decide which of the subsequent instructions in the datapath will fail their condition codes.

However, it is possible that the execution of a valid instruction which enters Ctrl3 and passes its condition codes will involve the modification of CPS. If the execution of such an instruction also generates a result to RReg, then it is not necessary to provide AddC with the new CPS; since RReg is unable to produce an acknowledgement message, Ctrl3 will block and thus no further instructions will be discarded by this process. For instructions which change the

CPS without producing a result though[3], it is essential explicitly to provide AddC with the updated CPS as Ctrl3 will be free to process and possibly discard more instructions, taking into account the new CPS.

In order to provide the new CPS to AddC, an alternative communication path between this process and Ctrl3 is required, for the existing one through WrtCtrl will be blocked. Hence, an extra path connecting the two processes is required in the model; since this communication is not included in AMULET1, buffering is necessary to decouple the processes and ensure that deadlocks due to their synchronization do not occur.

The interaction between AddC and Ctrl3 with regard to the CPS is illustrated in figure 9.7. A message with the current CPS will be issued by Ctrl3 for each valid instruction which follows an ILS_{Wch} instruction, until an instruction (valid or invalid) whose execution produces a result to be forwarded to RReg is encountered.

Typically, the number of consecutive valid instructions which follow an ILS_{Wch} without generating a result will be small. Therefore the communication overhead in occarm due to the extra messages sent to AddC by Ctrl3 will be insignificant.

Colour Mismatch

As described in section 6.5, instructions may be discarded due to a colour mismatch either in the ALU (Ctrl3 process) or in the primary decode unit (Decode1 process).

In order to determine the state of the datapath at any particular moment, it is essential for AddC to have information regarding the exact point where each of the invalid instructions is discarded. Such information is required by AddC only for instructions which immediately follow valid ILS_{Wch} instructions. This is necessary, as:

[3]These instructions are: TST, TEQ, CMP, CMN, MSR and MRS - see appendix A

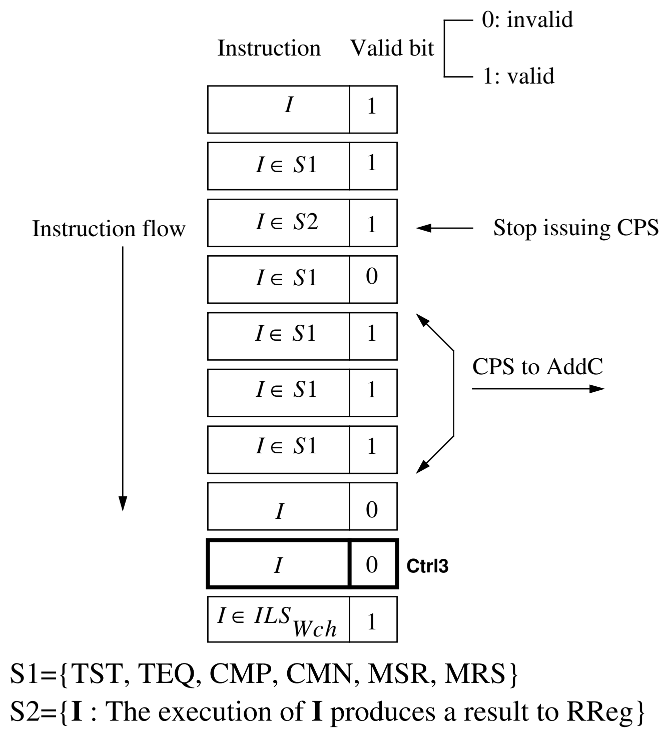


Figure 9.7: PDSP: Providing CPS to AddC

- Instructions which are discarded at Decode1 do not enter the datapath at all.
- Instructions which pass Decode1 may block the datapath before reaching Ctrl3 to be discarded (see sections 9.5.2.2-9.5.2.4) and,
- Instructions which do reach Ctrl3 to be discarded may still block the datapath if they produce a (invalid) result.

However, since the PCcol signal arrives at Decode1 in a nondeterministic fashion, the exact moment when instructions start being discarded in Decode1 is not directly available to AddC and thus must be explicitly provided by Decode1.

The most efficient way to provide a remote process (namely AddC) with a *single* piece of information (namely the point of arrival of the PCcol signal) is by means of a single message. However, such a scheme is not feasible in the occarm model. This message would have to be sent by Decode1 after the arrival of the PCcol signal; since this is nondeterministic, such is the generation of the message to AddC. Hence, if the blocking of AddC upon waiting for the message precedes the issuing of the message by Decode1, a deadlock may occur: the operation of the PC loop will stop and thus no more R15 values will be provided to Decode1 and eventually this process will block; if this happens before the PCcol has arrived the simulator will deadlock.

The alternative technique is to have Decode1 providing AddC with information regarding the outcome of colour checking, for each of the instructions involved, by issuing a separate message for each instruction. The generation of these messages commences as soon as Decode1 recognizes a valid ILS_{Wch} instruction which has the potential to activate PCcol (figure 9.8). AddC, in turn, blocks as soon as it detects that the ILS_{Wch} instruction has entered the datapath and thus no deadlock may occur. The transmission of the messages will stop as soon as the first instruction to be discarded by Decode1 is encountered or, in the case of the ILS_{Wch} instruction being invalidated in Ctrl3, upon arrival of the corresponding PCcol Null message from Ctrl3 (see section 9.6).

Similar to the communication between AddC and Ctrl3 regarding the CPS, an extra communication link with buffering is required to inform AddC of colour mismatches in Decode1.

9.5.2.2 Register Read Instructions

As explained in section 6.7, instructions attempting to read registers which are locked, will block until the registers are written and their addresses removed from the corresponding lock fifos; the blocking of read instructions is achieved by means of the register bank's read lock logic which, in occarm, is modelled by the ReadLock process (see figure 6.13). While a read instruction is blocked

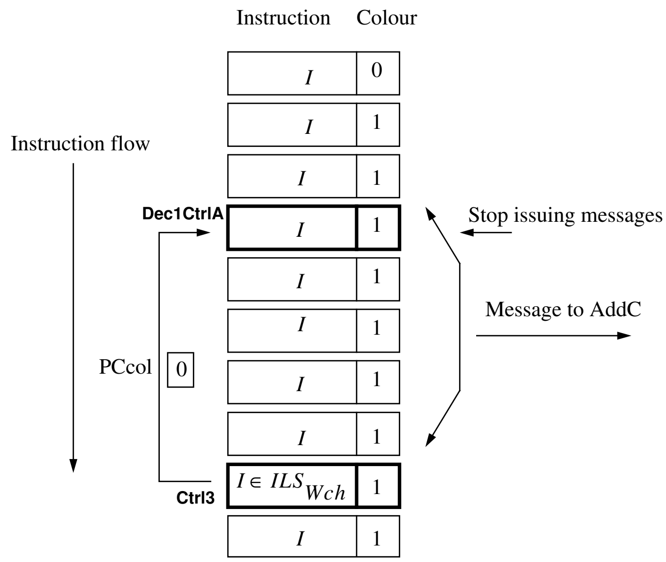


Figure 9.8: Informing AddC of Colour Mismatches at Decode1

and pending, the register bank will be unable to process any subsequent read operations; since every instruction which enters the datapath issues a read request to the register bank, instructions will backlog and the datapath will stall.

Write data are sent to the register bank through the write bus control logic (WrtCtrl process). However, during the execution of an ILS_{Wch} instruction, WrtCtrl remains blocked until the Wlx message from the AddC arrives; during this time, no further write data messages will be able to reach the register bank and unlock the corresponding registers. Thus, if the ILS_{Wch} instruction is followed by an instruction which attempts to read registers with pending writes, and the corresponding write data has not been forwarded to the register bank before WrtCtrl blocks, the datapath will stall imposing a limit to the number of consecutive PC messages that AddC may process during the execution of the PDSP algorithm.

Write data originate either from the datapath or the memory. If the write instruction which locks the register(s) follows the ILS_{Wch} in the datapath, the read instruction will definitely block at the register bank, for the result of the write instruction (write data or read memory address) will not be able to pass through WrtCtrl.

If the write instruction precedes the ILS_{Wch} however, and the write data originates from memory, then AddC can have no direct knowledge as to whether the data had been forwarded to the register bank before the blocking of the WrtCtrl took place; data and instructions from memory follow different paths (via DataIn and IPipe respectively, see section 6.4) and the order the respective messages arrive to WrtCtrl is non-deterministic. This knowledge however is possessed by WrtCtrl. Indeed, WrtCtrl may keep a record of the number of data values from memory that have been forwarded to the register bank since the last ILS_{Wch} instruction. This number is sent to AddC with the ILS_{Wch} message to be compared against the respective record, maintained by AddC, of the number of write operations preceding the current ILS_{Wch} instruction. Based on this information AddC may decide which of the write operations have been completed and thus which of the registers remain locked.

9.5.2.3 Instructions Activating the ALUgo Signal

The ALUgo signal is issued by Ctrl3 process to inform Decode1 of the validity of certain multicycle instructions[4]. It is activated by Ctrl3 as a response to the request message generated by Decode1 during the first cycle of the instruction. Decode1 blocks until the ALUgo message arrives; if it is positive (i.e. the instruction is valid), the next cycle of the instruction commences, otherwise the instruction is discarded so that it does not lock any registers (see section 6.6.2).

If the multicycle instruction follows an ILS_{Wch} instruction, and the datapath stalls before the request message from Decode1 reaches Ctrl3 ALUgo will not be issued and thus Decode1 will remain blocked.

[4]These instructions are: LDM, STM, any data processing instruction with Rd=R15 and S bit set (user mode), SWI and MSR that write to the CPSR (bit 22 (Pd) = 0).

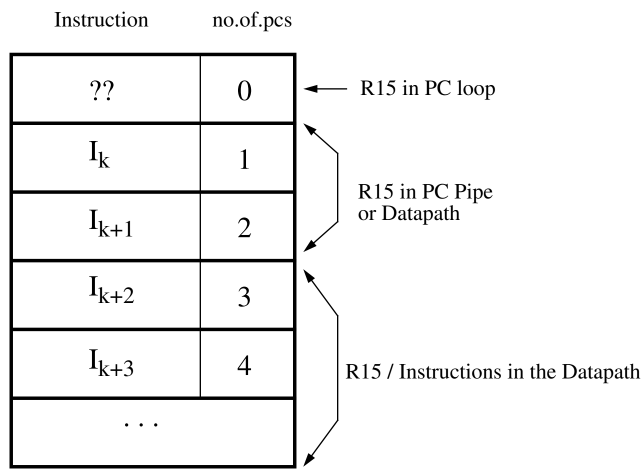


Figure 9.9: The Instruction Lookahead Table

Since AddC is informed of the state of the datapath at any particular moment, the decision as to whether the ALUgo signal is issued is straightforward.

9.5.2.4 Load/Store Multiple Instructions

As described in sections 6.6.2 and 6.3.1.3, during the execution of LDM/STM instructions Decode1 interacts directly with AddC via the LsmTrm channel (see figures 6.8 and 6.4). During this interaction, which starts upon the arrival at Decode1 of the corresponding ALUgo message and lasts until the operation of the LSM loop in AddC is complete, Decode1 will remain blocked not allowing any further instructions to enter the datapath.

Here too, the blocking of the datapath is directly detectable by AddC based on the information possessed by this process, and thus no further action is required.

9.5.2.5 The Instruction Lookahead Table

The main structure maintained by arbiter processes to make use of PDSP is the Instruction Lookahead Table (ILT). This may be implemented as a circular buffer and contains all the information required by the arbiter process to make decisions regarding the potential arrival of messages on its input links.

In the case of AddC, a new entry is appended to ILT each time a new instruction address (PC value) is let through to MAReg, to be forwarded to memory and the PC loop. Each entry of the ILT corresponds to an instruction and includes a field (*no.of.pcs*) that specifies the number of acknowledgement messages that have been received on the PCch channel after the corresponding instruction has been forwarded to MARreg (figure 9.9). For each new entry, *no.of.pcs* is initialized to 0; the existence of such an entry indicates that a message is expected on PCch. Each time a new PC value is let through to MAReg, $no.of.pcs$ of all entries in the ILT is incremented by one.

The *no.of.pcs* field provides information regarding the relevant position of instructions in the system. Indeed, since the size of the PC Pipe is two, if $no.of.pcs >= 3$, the instruction has definitely entered the datapath. If $no.of.pcs =$ 1 the instruction may be in the datapath or not (i.e. its R15 value may still be in the PC Pipe) depending on the number of preceding instructions. If $no.of.pcs = 2$, the instruction has entered Decode1 and the behaviour described in sections 9.5.2.3 - 9.5.2.4 is taken into account.

The Instruction Lookahead Table also enables AddC to predict future events on its Wch input link; an event is expected if ILT contains an entry with an ILS_{Wch} instruction.

9.5.3 The Wch Link

Messages arriving on Wch channel are primarily sent to AddC from the datapath (through WrtCtrl) as a result of the execution of ILS_{Wch} instructions and carry either branch target or data transfer addresses.

For data transfer operations whose destination register is R15, a second message will be sent over Wch, namely the new value of the Program Counter from memory.

According to the PDSP algorithm, when AddC detects an ILS_{Wch} instruction it blocks until it receives the corresponding messages on Wch. If however the ILS_{Wch} instruction will not be executed or the memory fails to respond (i.e. an abort occurs), the expected messages will never be issued, thus leaving AddC blocked and causing the simulator to deadlock.

There are two reasons why an instruction may not be executed, namely if its colour does not match that of the processor or if it fails its condition codes.

9.5.3.1 Colour Mismatch

All instructions whose execution may change the operating colour of the processor - either by explicitly writing a new value to the PC (i.e. the branch target address) or by causing an abort - belong to the Instruction Lookahead Set of the Wch channel. Thus, an ILS_{Wch} instruction will suffer a colour mismatch only if it follows another ILS_{Wch} which has changed the processor's colour.

For instructions which explicitly change the PC, the new colour is provided to the AddC with the branch target address. Since AddC is also informed of the colour of subsequent instructions entering the system as well as of instructions which enter the datapath, the decision as to whether an ILS_{Wch} instruction will be discarded is straightforward.

If however the colour changes due to an abort, AddC has no direct knowledge regarding this change; Ctrl3 will receive the abort signal from memory and will change the PCcol rejecting subsequent instructions. In this case a Null message must be sent by Ctrl3 to inform AddC of the occurrence of an abort and enable it to decide which of the following ILS_{Wch} instructions will be executed. If no such instruction exists, the Null message is simply ignored by AddC; however it still needs to be issued since Ctrl3 can have no knowledge regarding the existence of subsequent ILS_{Wch} instructions in the datapath or their possible rejection by

Decode1[5].

Thus, for ILS_{Wch} instructions which may abort, AddC waits for two messages on Wch link, namely the memory address and a message regarding the abort status; if R15 is included in the destination register list of the instruction, the second message expected by AddC will be either the new value of the PC arriving from memory or, in the case of abort, a Null message from Ctrl3.

9.5.3.2 Condition Codes Failure

As described in section 9.5.2.1, when a valid ILS_{Wch} instruction which does not activate the abort signal is executed, AddC is provided by Ctrl3 with all the CPSR-related information required to predict the fate of subsequent valid instructions regarding their condition codes; this is performed via a dedicated path and lasts until a result is produced by Ctrl3 and forwarded to RReg register.

The generation of the result to the RReg register has a dual effect:

- Firstly, as explained in section 9.5.2.1, initially it blocks Ctrl3 preventing it from receiving any more instructions and thus eliminating the need for further informing AddC of possible changes in the CPSR. This role is performed for as long as RReg is unable to produce an acknowledgment message to Ctrl3.
- Secondly, it informs Ctrl3 that the last address message produced by this process has been selected and processed by AddC. Indeed, the acknowledgement message from RReg will not be issued until this register receives an acknowledgement from WrtCtrl; this in turn will be generated upon receipt of the Wlx signal.

[5]The Null message could be sent by Ctrl3 upon the arrival of the first ILS_{Wch} instruction after the abort. If however no such instruction reached Ctrl3 (i.e. they are all discarded in Decode1) then the Null message would never be sent and AddC would remain blocked.

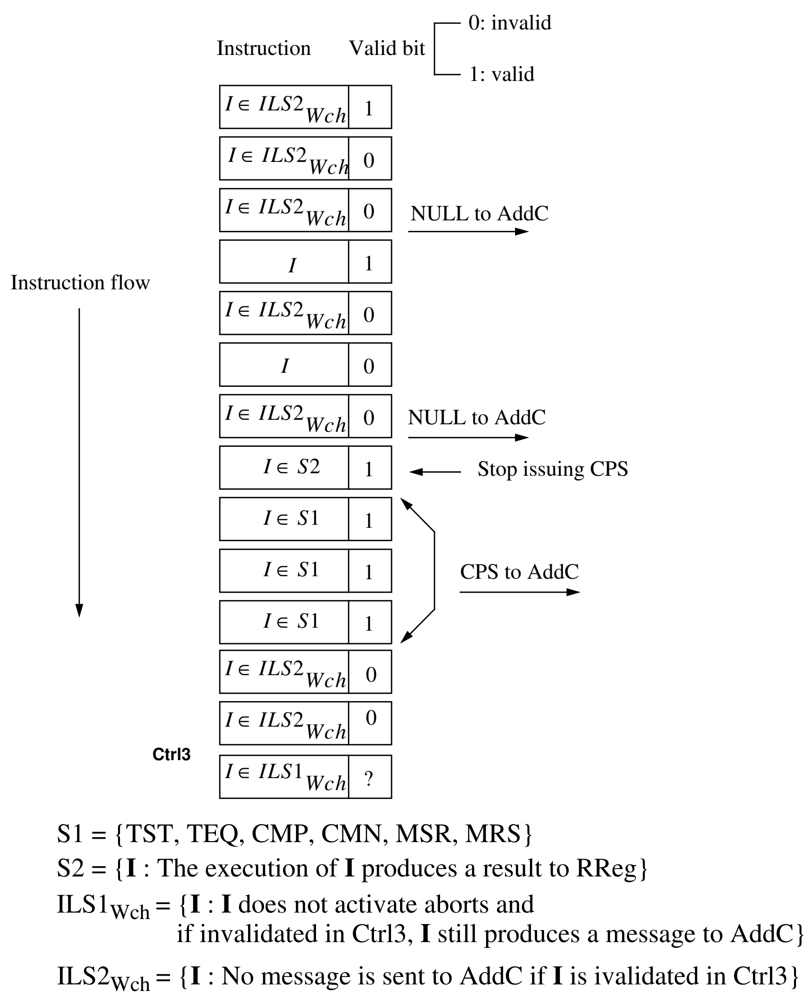


Figure 9.10: PDSP messages from Ctrl3 to AddC

Once the acknowledgement from RReg is received, no further CPSR-related messages are sent to AddC. Thus, for subsequent ILS_{Wch} instructions which fail their condition codes, Null messages are required to be sent by Ctrl3 to inform AddC of this event. These messages may be redundant when they refer to instructions for which AddC may already have been informed of their fate (i.e. instructions reaching Ctrl3 immediately after the receipt of the acknowledgement from RReg); however Null messages are still required as Ctrl3 can have no knowledge as to which these instructions might be. Once a Null message is sent to AddC, no more messages of this kind will be issued for ILS_{Wch} instructions subsequently invalidated in Ctrl3, until a valid instruction is executed.

This pattern will be followed until the next ILS_{Wch} to produce a result is encountered, whereupon the production of CPSR-related messages to AddC as described in section 9.5.2.1 will commence. This is illustrated in figure 9.10 where a complete picture if the interaction between Ctrl3 and AddC is provided.

It is interesting to note that when Null messages are produced by Ctrl3, the path leading to AddC via Wch is unblocked and thus these messages will be forwarded without affecting the behaviour of the system by filling registers that would be otherwise occupied by system's messages.

9.6 The Primary Decode Arbiter

As described in section 6.6.1, arbitration is employed by primary decode (Decode1) to detect the PCcol signal from the ALU. Accurate modelling of this arbiter in occarm would ensure that the number of instructions discarded as invalid by the Dec1CtrlA process is equal to the respective number in AMULET1. PCcol is activated each time the processor's colour changes as a result of instructions which modify R15 or abort. Thus the Instruction Lookahead Set of the PCcol channel is:

$\mathbf{ILS}_{PCcol}$=\{*B, BL, SWI, LDR, STR, LDM, STM, Data Processing with PC as Dest. Reg.*\}

which is the same as the ILS_{Wch}.

As the Dec1CtrlA process constitutes the datapath's entry point, it has immediate knowledge of the instructions entering the system and consequently of the potential of the PCcol to be activated. Thus, the application of Rule 1 of the PDSP (section 9.3.2) is straightforward.

However, it is essential to ensure that the blocking of Dec1CtrlA does not affect the operation of the AddC process to avoid deadlocks. This is due to the fact that $ILS_{PCcol} = ILS_{Wch}$ and Dec1CtrlA, Ctrl3 and AddC processes are connected together in a closed loop (see figure 9.6); if Dec1CtrlA blocks while Ctrl3 is waiting to interact with AddC before it activates the PCcol signal, and AddC expects an acknowledgement from Dec1CtrlA (through the PC Pipe), PCcol will never be issued and occarm will deadlock.

For branches, software interrupts and data processing instructions, the PCcol message is issued by Ctrl3 in parallel with the new PC value (branch target address) sent to AddC. Thus, since there is no interdependency between the interactions of Ctrl3 with Dec1CtrlA and AddC, the blocking of Dec1CtrlA will not lead to deadlock. If the instruction fails its condition codes, a Null message is sent by Ctrl3 over PCcol channel; Dec1CtrlA will not block until the next valid ILS_{PCcol} instruction is encountered.

For instructions which cause a colour change only if they abort however, PCcol will not be issued until after Ctrl3 has received the abort signals from memory. Thus, to avoid deadlocks, Dec1CtrlA should not block until the data address which is produced as a result of the $ILS_{PCcol/Wch}$ instruction, has been processed by AddC and forwarded to memory.

This situation is illustrated in figure 9.11, where I_1 is an ILS_{PCcol} instruction which may abort (e.g. LDR). The execution of this instruction results in a message W_{I_1} being sent to AddC with timestamp $t_{W_{I_1}}$. If Dec1CtrlA blocks waiting for PCcol, and no acknowledgement from the PC Pipe with timestamp $t_{Ack_k} > t_{W_{I_1}}$ is encountered by AddC, the PC Pipe will stall and the system will deadlock.

Since Dec1CtrlA has no immediate knowledge as to when the PC loop is interrupted (i.e. an acknowledgement with timestamp $t_{Ack_k} > t_{W_{I_1}}$ is encountered), a technique needs to be devised to provide it with this information.

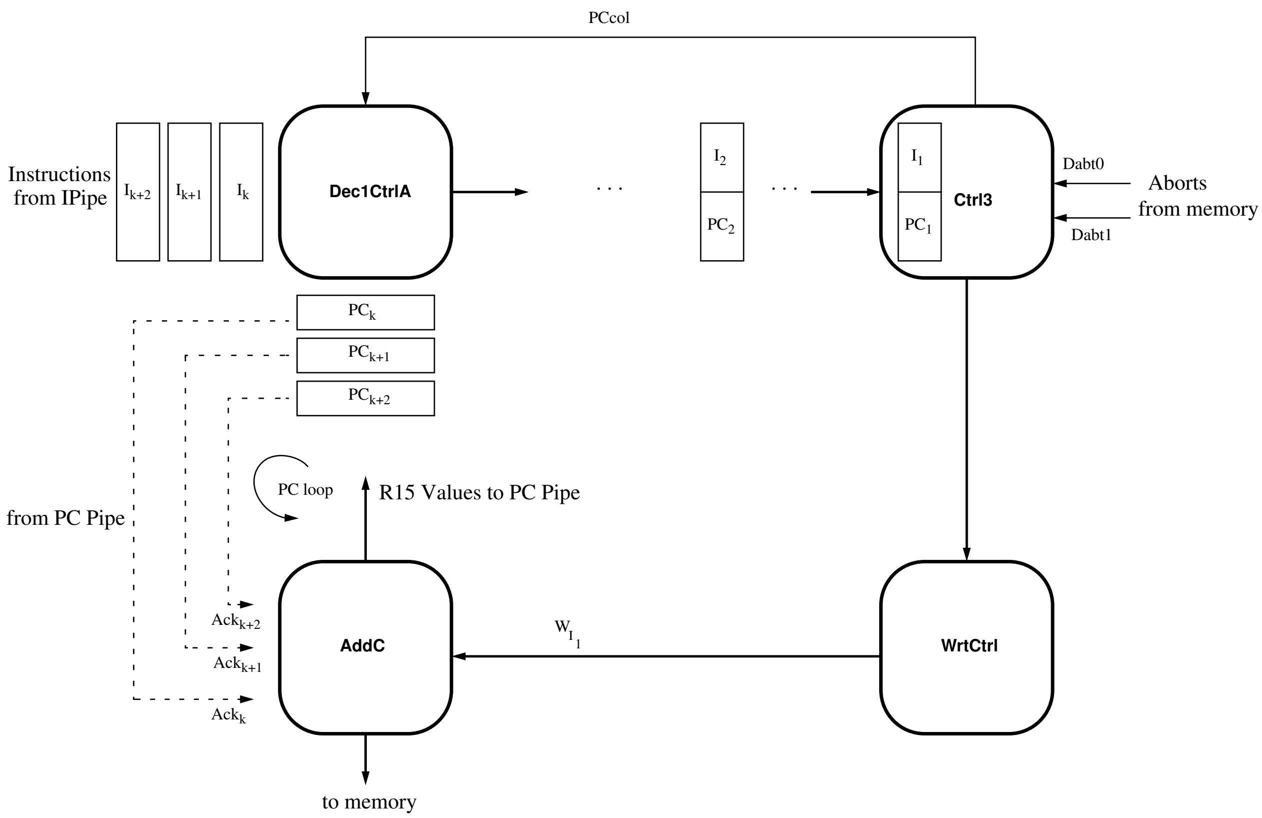


Figure 9.11: The Decode1 Arbiter: PCcol due to Aborts

A possible approach would be to provide this information with the PC value forwarded by AddC immediately after the interrupt, i.e with PC_{k+1} (figure 9.11). This technique would not involve any extra communication overhead, however, it does not guarantee the prevention of preemptions. Indeed, if $t_{I_k} > t_{PCcol}$, then by the time Dec1CtrlA receives PC_{k+1} a preemption will have already occured. Thus, it is essential to provide Dec1CtrlA with information regarding the fate of Ack_k before the entry of I_k into the datapath takes place.

This may be achieved by means of an extra (buffered) link directly connecting AddC and Dec1CtrlA as depicted in figure 9.12. Each time AddC selects the next address to be forwarded to memory, it issues a message to inform Dec1CtrlA of the selection's result; the production of messages commences as soon as AddC detects an ILS_{PCcol} instruction and stops upon selection of the address message on Wch

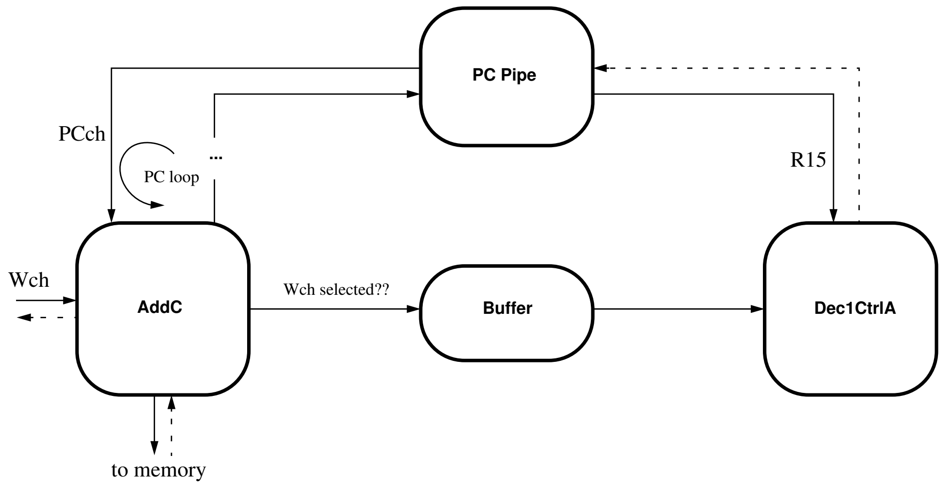


Figure 9.12: Informing Decode1 of the selected channel

channel. Similarly, for each instruction received after an ILS_{PCcol}, Dec1CtrlA also waits for the corresponding message from AddC before it decides whether it should let the instruction enter the datapath or block and wait for PCcol.

It is important to note that the size of the instruction pipe (IPipe) guarantees that once a data address is forwarded by AddC to MAReg it will reach memory irrespective of the state of the datapath [Pave94] pp. 128-131. Thus, blocking Dec1CtrlA has no effect on the behaviour of the system and, consequently, the issuing of the abort signals by memory.

If the ILS_{PCcol} instruction does not abort, a Null message is sent over the PCcol channel by Ctrl3. If the instruction fails its condition codes, Dec1CtrlA is informed by means of a Null message sent from AddC over the extra link. This is due to the fact that Dec1CtrlA will have to receive an unknown number of messages issued on the extra link before AddC became aware that the the ILS_{PCcol} instruction was discarded.

For load/store multiple instructions no special messages are required to be sent from AddC, for no more instructions will enter the datapath until the load/store multiple operation is complete; the Dec1CtrlA process will block and wait for

PCcol as soon as its interaction with RdGen process completes (see section 6.6.2).

9.7 The Write Control Arbiter

Within the write control unit of AMULET1, arbitration is used to enable data values originating from either the datapath or the memory to gain access to the write bus (see section 6.9); in occarm, an ALT statement is employed to provide arbitration on the corresponding channels, namely DPch and DINch respectively (see figure 6.19). The Instruction Lookahead Sets of these channels are:

$\mathbf{ILS}_{DPch}$=$\{$*B, BL, SWI, LDR, STR, LDM, STM, Data Processing*$\}$

$\mathbf{ILS}_{DINch}$=$\{$*LDR, LDM*$\}$

9.7.1 The DINch Link

DINch channel will fire as a result of a message's having previously been sent to WrtCtrl process over DPch channel. Indeed, addresses of data to be loaded from memory are forwarded from the datapath (Ctrl3) to the address interface (AddC) via WrtCtrl. Thus the prediction as to whether a data value is expected from memory is straightforward.

For an LDR instruction, a single message will be issued from memory. If the instruction aborts, a data value, albeit invalid, will still arrive on DINch channel so there is no need for extra Null messages.

In the case of load multiple operations however (LDM), more than one message will be issued from memory; the number of these messages will be equal to the number of registers to be loaded. According to the AMULET1 specification, during the first cycle of the LDM instruction, Ctrl3 process outputs the base address and then blocks until the abort/no abort signal arrives from memory [Pave94], pp. 86-90; hence during this time *no* message will be sent over the

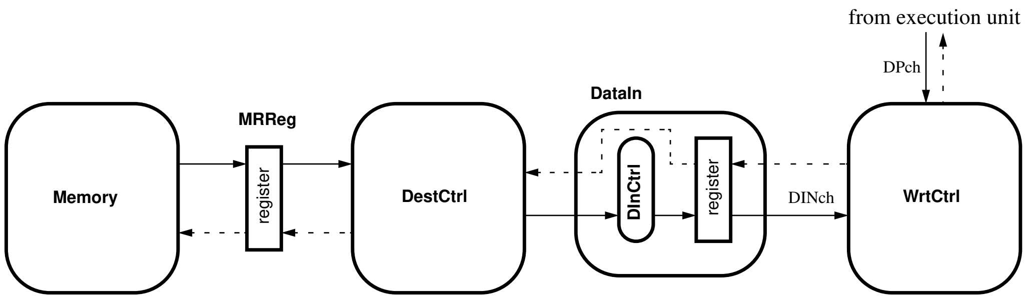


Figure 9.13: The Memory-WrtCtrl pipeline

DPch channel[6] and thus WrtCtrl will continuously accept and process messages arriving on DINch.

The abort/no abort signal is issued as soon as the last transfer address arrives at memory, whereupon Ctrl3 is free to proceed with its operation executing further instructions and producing results on the DPch channel. Consequently, as soon as the abort/no abort signal is issued, the continuous processing of messages from DINch by WrtCtrl must stop otherwise preemptions might occur. WrtCtrl should accept and process sufficient data messages to allow the *last* data address to reach memory.

The pipeline between the memory and WrtCtrl consists of two registers, namely MRRreg and DataIn (figure 9.13). Hence, if N is the number of registers to be loaded, then the last address will reach memory as soon as WrtCtrl processes the (N-2)th message from DINch; it is at this point that the abort signal will be activated to free Ctrl3. Any attempt to block WrtCtrl before the processing of the (N-2)th message has completed will result to deadlock as the abort will never be issued while processing (N-1)th message before taking into account DPch channel may result into a preemption.

The functionality of WrtCtrl with regard to DINch link is depicted in figure 9.14.

[6]Actually, if writeback is specified, one more message will be sent from Ctrl3 on DPch channel.

```
PROC PDSP.WrtCtrl()
  SEQ
    ...
    --DINch.selected=TRUE
    IF
      instruction=LDM
        SEQ i=1 FOR N-2
          SEQ
            DINch?message
            --process and forward message
      instruction=LDR
        SEQ
          DINch?message
    --continue normal PDSP operation
    ...
:
```

Figure 9.14: WrtCtrl: Reading data values from memory

9.7.2 The DPch Link

As explained in the previous section, in AMULET1 there is a regular flow of messages arriving at the write control from the datapath, while messages from memory will be issued only occasionally, as a result of the former.

Consequently, the WrtCtrl process will normally be ready to accept messages from the DPch channel. However, there is a possibility that such messages will never arrive and as a result the system will deadlock. This may happen only if there are data messages from memory pending, and may be due to:

- The register bank being stalled as a result of an instruction that attempts to read a locked register which is to be written by one of the pending messages on the DINch link.
- The emptying of the datapath. This may occur if no more instructions enter the datapath due to the following:
  1. The address interface stops issuing instruction addresses to memory. As soon as an address whose contents are to be written to R15 is issued to memory[7], the operation of the PC loop stops until the corresponding

[7] In the case of an LDM, the operation of the PC loop will stop upon issuing to memory the

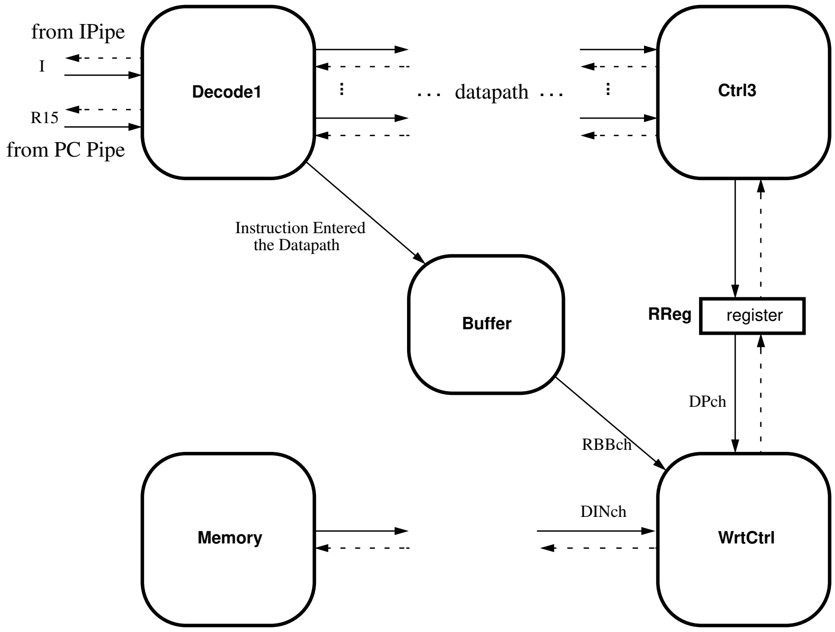


Figure 9.15: Bypassing the register bank

data arrives from memory (through Write control) to become the new circulating value of the PC loop.

2. Instructions issued from memory are prevented from reaching the instruction pipe (IPipe). Both instructions and data values from memory enter the processor via the MRReg register. If the number of pending data values from memory is equal to the number of registers in the pipeline which connects the memory to the WrtCtrl (i.e. two, namely MRRreg and DataIn, see figure 9.13) then the following instructions will be unable to enter MRReg and consequently to reach Decode1.

In order to resolve the first of the aforementioned issues, namely the stalling of the register bank, the WrtCtrl process needs to have knowledge regarding the instructions which have entered the datapath. A possible technique to provide WrtCtrl with this information is depicted in figure 9.15. An extra buffered link (namely, the *RBBch*) has been introduced whereby Decode1 (DecCtrlA) sends a

base address.

```
PROC PDSP.WrtCtrl()
  SEQ
    ...
    --DPch.expected=TRUE
    PRI ALT
      DPch?result.message    --from Ctrl3
        SEQ
          --process and forward message
          --if corresponding entry exists in ILT, delete it
      RBB?instruction        --from Decode1
        SEQ
          IF
            corresponding.message.from.DPch.already.processed
              SEQ
                --ignore instruction
            TRUE
              SEQ
                --append entry in ILT of DPch channel.
    ...
:
```

Figure 9.16: WrtCtrl: Reading messages from the datapath

message to WrtCtrl each time a new valid instruction enters the datapath.

The functionality of WrtCtrl regarding the acceptance of messages from the datapath is depicted in figure 9.16. An ALT statement enables WrtCtrl to read messages arriving either from Decode1 (new instructions) or the Ctrl3 (produced as a result the execution of ILS_{DPch} instructions). If a message regarding the entry of a particular ILS_{DPch} instruction into the datapath is received by WrtCtrl *before* the corresponding result from Ctrl3, then an entry is appended to the Instruction Lookahead Table of the DPch channel; otherwise it is simply ignored.

This scheme, whereby messages from Ctrl3 may be received *before* WrtCtrl is informed of their potential arrival ensures a higher degree of parallelism in the model. As WrtCtrl will receive and process messages from Ctrl3 as soon as they become available, without it having to block first for the corresponding messages from Decode1, the PRI ALT ensures that messages from Ctrl3, which are crucial as they form part of the operation of the simulated system, are treated with higher priority.

For ILS_{DPch} instructions which are discarded in Ctrl3, a Null message is issued by this process to inform WrtCtrl of this event.

Regarding the emptying of the datapath due to the loading of R15 (case 1 above), WrtCtrl needs to be informed that no more instructions are going to enter the datapath before the corresponding message from DINch channel is processed and forwarded to AddC. This may be achieved by having the AddC process issue a Null message to Dec1CtrlA immediately after the operation of the PC loop stops; this message is then forwarded directly to WrtCtrl by Dec1CtrlA. It is important to note that the Null message *will* reach Dec1CtrlA; after the PC loop interruption the PC Pipe will become empty thus ensuring the propagation of the Null message.

This mechanism can not be employed to deal with the blocking of MRReg (case 2 above) as AddC has no knowledge regarding the number of pending data values. Furthermore, the PC Pipe will be occupied by PC values pending the pairing with the corresponding instructions which will not arrive until MRReg is freed; thus no information can reach Dec1CtrlA through this path.

AddC however possess a record (in the ILT) of all the instructions which have been sent to the instruction pipe immediately *before* the last data address was issued to memory. Indeed, the acknowledgement message which will be issued by the MAReg in response to its receiving the data address will carry the last instruction which has been issued by memory; if the corresponding data value blocks the MRReg, no more instructions will reach the instruction pipe.

This information may be made available to the WrtCtrl process by means of a single message, namely the Wlx signal which is issued as soon as the acknowledgement from the MAReg is received. Based on this information, WrtCtrl is able to decide the maximum number of messages it should expect from the datapath.

Model	Elapsed Time (minutes)
$Occarm_{ALT}$	1.72
$Occarm_{PDSP}$	2.15
$Occarm_{PDSP-MLL}$	1.93

Benchmark: Dhrystone (1 loop)

Table 9.2: Performance of PDSP (Address Interface)

Not all the instructions included in the ILT of AddC will enter the datapath however; some may be discarded as invalid at Dec1CtrlA. Therefore, in order to avoid deadlocks that might occur as a result of WrtCtrl waiting for discarded instructions, a Null message is sent by Dec1CtrlA upon receipt of PCcol to inform WrtCtrl of the point, after which invalid instructions will be discarded.

9.8 Performance Evaluation of PDSP

In order to get an indication of the potential impact of PDSP to the performance of the simulator, the algorithms described in section 9.5 have been implemented and incorporated into occarm to develop an accurate model of AddC process.

The performance results obtained are illustrated in table 9.2. The application of PDSP onto the AddC arbiter process of the occarm model, when no attempt is made to exploit the Minimum Latency Lookahead (i.e. using the algorithm of figure 9.1). has increased the time required by the model to execute one Dhrystone loop by 0.43 minutes, thus resulting in a 25% decrease of the performance of the simulator.

Some preliminary experiments have indicated that by exploiting MLL a performance improvement of at least 10% can be achieved, reducing the time required for the accurate execution of one Dhrystone loop by 0.22 mins.

Work is still to be done to implement and incorporate into occarm all the algorithms described in this chapter so that a detailed performance evaluation and confirmation of correctness of PDSP is possible.

9.9 Summary

This chapter has presented the Program Driven Synchronization Protocol, a novel approach for dealing with the causality problems introduced by the distributed nature of the proposed modelling philosophy. This is a conservative, deadlock avoidance approach whose objective is to exploit the ability of arbiter processes to predict events in the simulated future based on the instructions executed in the model. The feasibility and robustness of this approach has been demonstrated by applying its concepts to the occarm simulation model.

Chapter 10

Conclusions and Further Work

"Τὶ φὴς; Νεφὲλης αρ' ἀλλως εὶχομεν πὸνους περὶ;"

Ευριπὶδης, Ελὲνη.

"What are you saying? It was only for a cloud that we struggled so much?"

Euripides, Helen.

10.1 Background

Synchronous VLSI design is approaching a critical point, with clock distribution becoming an increasingly costly and complicated issue and power consumption rapidly emerging as a major concern. Asynchronous digital design styles promise to liberate VLSI systems from clock skew problems, offer the potential for low power and high performance and encourage a modular design philosophy which makes incremental technological migration a much easier task.

The desire to exploit the potential advantages offered by asynchronous logic has recently fueled a revival of interest in asynchronous systems. Micropipelines, one of several design techniques that have been proposed, offer a good engineering framework for the design of asynchronous systems. AMULET1, the first asynchronous Micropipelined implementation of a commercial RISC processor, has demonstrated the feasibility of employing asynchronous logic to build large and complex systems.

Modelling and simulation, being at the heart of digital system design, may perform a catalytic role in the quest for the realization of the potential offered by asynchronous logic. Hence, the recurrence of interest in asynchronous design has been accompanied by intense research activity aimed at developing notations and techniques appropriate for modelling and simulating asynchronous systems.

The concurrent, asynchronous, process-based model of computation of CSP, with the support for non-deterministic behaviour, and the point-to-point, synchronous and unbuffered inter-process communication are particularly suitable for describing the concurrent, non-deterministic behaviour of asynchronous hardware systems and provide a natural and convenient means for the rapid construction of asynchronous hardware models. Hence, CSP and occam have long and extensively been advocated as potential notations for the description of asynchronous hardware and various CSP-based and occam-like notations have already been employed for this purpose. However all the work undertaken so far in this area, has placed emphasis on producing specifications to be used as input to silicon compilers for the automatic synthesis of asynchronous circuits. As a consequence, the notations developed, and the modelling approaches used have been intended to match the particular approach employed for the silicon compilation process.

However, if the benefits of using CSP for describing asynchronous systems are to be exploited and taken advantage of, it is essential to use a standard specification language, that would be easily and widely available.

Occam may well serve this purpose: it is based on CSP, it is an executable programming language with well defined syntax and semantics, it is widely used and commercially supported, and is expected to be supported by a wide range of hardware platforms [Ofal]. Furthermore, occam is a parallel language which may be executed on multiprocessor systems and thus has the potential for high performance.

The goal of the research presented in this thesis was to investigate the suitability of occam for the modelling and simulation of complex asynchronous hardware systems.

10.2 Contribution of the Thesis

The research presented in this thesis is the first one to investigate the use of the standard occam language for the description of asynchronous, micropipelined architectures. There were a number of challenges presented by this endeavour:

- A modelling approach had to be developed, which would provide the framework for the construction of occam models.
- A number of important problems related to the execution of occam models needed to be resolved.

The research presented in this thesis has addressed all these issues.

10.2.1 Modelling

A factor which is crucial for the exploitation of the advantages offered by CSP is the availability of a consistent and generic modelling framework. This thesis has introduced such a framework. This framework exploits the connection between the semantics of occam and the behaviour of asynchronous hardware systems as well as the parallelism inherent in asynchronous hardware to facilitate the development of parallel architectural simulation models. Within this framework, the system is modelled as a network of concurrent, communicating, data-driven occam processes, with each process modelling either a register or a piece of control circuitry; a generic register model has been introduced for the first time, which accurately and faithfully represents the actions and parallel behaviour of event-driven registers. The suitability and robustness of the modelling framework introduced in this thesis has been demonstrated by building occarm, an occam model of the AMULET1 asynchronous microprocessor.

10.2.2 Simulation

An important issue, which the advocates of CSP and occam, as modelling notations for asynchronous systems, have largely overlooked and neglected, and which is addressed for the first time by the research presented in this thesis, is simulation, namely the execution of the occam model on a computer system. The research presented in this thesis has concluded that there are two major issues that need to be taken into account, if an occam model is to be used for simulation:

- The exploitation of the relationship between CSP/occam and asynchronous hardware systems, implies a data-driven operation of the processes of the model. The execution of a model consisting of data-driven processes which describe the modelled hardware at the Register Transfer (or higher) level is straightforward. This however is not true for models at lower levels of abstraction. One can certainly use CSP/occam to produce textual descriptions of gates and event processing blocks. However, the data-driven operation of an occam model whose concurrent processes model gates and event processing blocks with level sensitive inputs (i.e. Select) may lead to deadlocks or incorrect results. In this case, the simulated time is needed to act as the synchronization agent in the model; this results in a conventional, discrete event simulation activity where the relationship between CSP/occam and asynchronous hardware is no longer the basis of the operation of the model.

- Even at the Register Transfer, or higher, level, the exploitation of the relationship between CSP/occam and asynchronous hardware systems, trades temporal accuracy for ease of modelling. Although the topological characteristics of the modelled hardware system map naturally onto the model, the temporal characteristics do not, a situation that may lead to causality errors in the model. This is a problem inherent in any decentralized, distributed simulation approach. In occam architectural models, causality errors are a consequence of adopting a data-driven approach to model arbiters, using the occam ALT construct. Although, causality errors have no impact on the correct operation of the model, they introduce an error in the simulated time and consequently, affect the accuracy of the evaluation of the simulated system. The research presented in this thesis has a dual contribution to this area:
  - Firstly, it has attempted to quantify the inaccuracy introduced by causality errors, using the occam model of the AMULET1 microprocessor as a testbed.
  - Secondly, it has introduced the Program Driven Synchronization Protocol, a novel synchronization protocol which attempts to eliminate causality errors, within the proposed modelling framework.

In his PhD thesis, Brunvand mentions that occam-like “programs written to be translated into a circuit may just as well be compiled into object code and executed on ordinary computers. The system may then be simulated by running the compiled program like any other program” [Brun91]; the execution of an occam model on a computer system however is a much more complicated endeavour. The distributed nature of occam and the characteristics of the development tools associated with the language impose a number of issues that need to be resolved if the model is to serve as a simulation tool rather that just a simple textual description. These issues include debugging, monitoring and terminating the simulation; for distributed, multi-processor implementations, mapping and load balancing issues also need to be considered. Within the research presented in this thesis, a number of techniques and mechanisms have been developed which address all the aforementioned issues in the context of occam asynchronous architectural models.

10.3 The Program Driven Synchronization Protocol

Addressing the causality problems introduced by the distributed nature of the proposed modelling philosophy, this thesis has introduced the Program Driven Synchronization Protocol (PDSP). This is a general theoretical framework for the development of accurate arbiter processes which eliminate preemptions, while preserving the data-driven philosophy of the modelling approach.

PDSP is based on the conservative, deadlock avoidance approach. It exploits the characteristics of the simulated architectures to enable arbiter processes predict their simulated future based on the instructions executed by the model; the term "Instruction Lookahead" has been introduced to refer to this concept. The exploitation of Instruction Lookahead ensures that Null messages are issued only when necessary, and are directed only to arbiter processes. This philosophy constitutes a significant departure from the conventional, Chandy/Misra/Bryant-based, conservative, deadlock avoidance protocols where *all* processes in the model are required to handle and generate a regular flow of Null messages throughout the model.

The success of PDSP depends on the ability to exploit the "Instruction Lookahead" properties of the simulated architecture. The application of the PDSP to the occarm simulation model has demonstrated that such an exploitation is feasible, even for systems of the AMULET1's complexity.

The work with occarm has demonstrated that the application of PDSP may require the development of appropriate algorithms to provide arbiter processes with the necessary knowledge and enable them to make decisions regarding the arrival of messages on their input channels. The design of such algorithms requires an extensive knowledge of the operation of the simulated architecture. This may be considered the major drawback of PDSP; however this is a problem typical of the conservative framework and not specific to PDSP. The algorithms per se are simple to implement and their impact on the model's philosophy and complexity is minimal.

10.4 Performance

The simulation of digital systems in general, and computer architectures in particular, has long been categorized among the highly computation intensive applications. The same is true for the simulation of asynchronous digital systems. For the testing and evaluation of the AMULET1 design, for instance, more than 4 million instruction cycles were simulated [Pave94], a number which corresponds to many hours of simulation. Hence, a parallel approach to simulation, such as the one described in this thesis, could contribute significantly in reducing the duration and cost of the design cycle. However, the performance achieved by the multi-processor implementation of occarm is far from satisfactory. This deficiency has been attributed to the particular characteristics of the testbed architecture (AMULET1) and the transputer technology used for the research described in this thesis.

The asynchronous architectures currently under development are generally characterized by a higher degree of parallelism and more regular interconnection patterns than AMULET1. These characteristics coupled with the communication efficiency promised by the T9000 transputer will possibly allow the potential for high performance offered by the modelling approach presented in this thesis to be realized.

10.5 Occam as an Asynchronous Hardware Description Language

CSP-based parallel languages, with the static, process-based model of computation they support, provide a natural and convenient means for the expression of the behaviour and structure of asynchronous computer systems. Occam in particular, with its support for explicit control of concurrency even at the command level, and its simple "send" and "receive" commands is particularly suitable for describing digital systems.

Occam can describe asynchronous control circuits at a level which is very close to their implementation; consequently it may provide guidance for the realization of the design (e.g. an IF statement will correspond to a Select block, a PAR of input commands will be implemented using a Muller-C block etc). This characteristic may also be exploited for the automatic derivation of circuits from occam specifications as suggested in the next section.

Furthermore, the parallel, distributed nature of occam forces the designer to think in "asynchronous terms" and to perceive its design as a distributed, asynchronous structure where a global state does not exist. The work with occarm has suggested that this may be the most important advantage of using occam for asynchronous architectural modelling. Indeed, the construction of occarm exposed behaviour patterns of AMULET1, thitherto unknown (e.g. the behaviour of the PCcol signal), whose operation was time-dependent rather than asynchronous.

On the negative side, occam lacks several data structures (and protocols for that matter) that would be extremely useful in a distributed simulation endeavour (e.g. records). Furthermore, for the manipulation of hardware circuit models, efficient and easy means for treating numbers as booleans (and the reverse) are extremely important, an area where occam is weak.

Another disadvantage of occam, as suggested by the programming effort of the research presented in this thesis, is its rigid and verbose layout format (reminiscent of FORTRAN) and the semantic significance of indentation which makes both, the development and debugging of programs time consuming and frustrating tasks.

Occam2.1 and the new development system in support of the T9000 transputer alleviate some of the aforementioned deficiencies (e.g. records).

10.6 Further Work

The research presented in this thesis suggests a number of areas where further research and development work may be undertaken.

10.6.1 Modelling and Simulation

The work described in this thesis was based on a particular case example, namely the AMULET1 architecture. The ideas and techniques produced as a result of this work should be applied on different asynchronous designs so that their validity and generality may be established.

In particular, the feasibility and applicability of PDSP as a general synchronization protocol should be examined by employing it to devise preemption free arbitration algorithms in more existing asynchronous architectures. Furthermore, the remaining algorithms, described in chapter 9, should be implemented to allow a more extensive performance evaluation of PDSP to be performed. Work in this direction has already commenced.

Tools for the postmortem analysis of the monitoring data would also be particularly desirable. The analysis could be indented to the generation of graphical representations of the system's behaviour (e.g. waveforms) or the automatic detection of deadlocks based on the collected traces of parallelity events.

10.6.2 Automatic Synthesis

CSP-type specifications are being increasingly used for the automatic synthesis of digital circuits. Occam may also be used for this purpose. Already, subsets of occam and occam-like languages have been used for the automatic synthesis of both synchronous [Page91] and asynchronous [Brun91] circuits. A silicon compiler could be developed, which would accept, as input, RTL asynchronous architectural occam models, built following the approach presented in this thesis, and automatically generate the equivalent gate-level circuit descriptions using syntax directed translation. The output of the compiler could be a netlist which could then be simulated (using a sequential albeit time-accurate discrete event simulation) or fed into a CAD tool to be implemented.

Appendix A

The ARM6 Programmer's Model

This appendix provides a short description of the programming model of the ARM6 processor. A more detailed description of the processor and its programming may be found in [Furb89] [VLSI90]. The information presented in this appendix is based on that given in [Furb89] [Furb95] and [Pave94]; figures A.1 and A.2 have been kindly provided by N. Paver.

ARM6 is a 32-bit RISC processor. The data types supported by the processor are bytes (8 bits) and words (32 bits) which are transferred by means of a 32-bit data bus; memory addresses make use of a separate 32-bit address bus.

ARM6 supports six modes of operation, namely *USER* (for normal program execution), *FIQ* and *IRQ* (for dealing with interrupt handling), *SVC* (protected supervisor for the operating system) and *ABT* (for dealing with data and instruction prefetch aborts) and *UND* (entered when an undefined instruction is executed).

A.1 The Registers

The ARM6 processor has a total of 37 registers consisting of 31 general purpose 32-bit registers and 6 status registers.

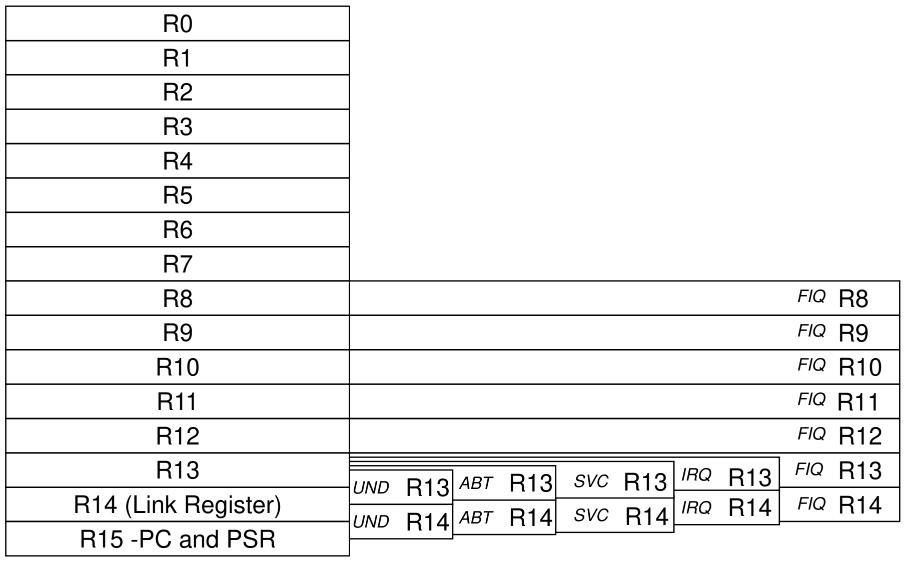


Figure A.1: The ARM6 Register Organization

The 31 general purpose registers are organized into a set of partially overlapping banks as depicted in figure A.1. At any time, 16 general purpose registers are visible to the programmer, namely R0 - R15. The visible registers depend on the processor mode while the rest (referred to as *banked registers*) are switched on to support IRQ, FIQ, SVC, ABT and UND modes. R15 contains both the program counter and the program status register, as illustrated in figure A.2. R14 is used as the subroutine link register and receives a copy of R15 when a Branch and Link instruction is executed.

The status registers consist of the Current Processor Status Register (CPSR), which is visible in every mode, and a set of Saved Processor Status Registers (SPSRs), one for each of the non-user modes (figure A.3).

A.2 The Instruction Set

The ARM instruction set is based on the load/store model; no memory to memory operations are supported. Instructions are exactly one word and data operations

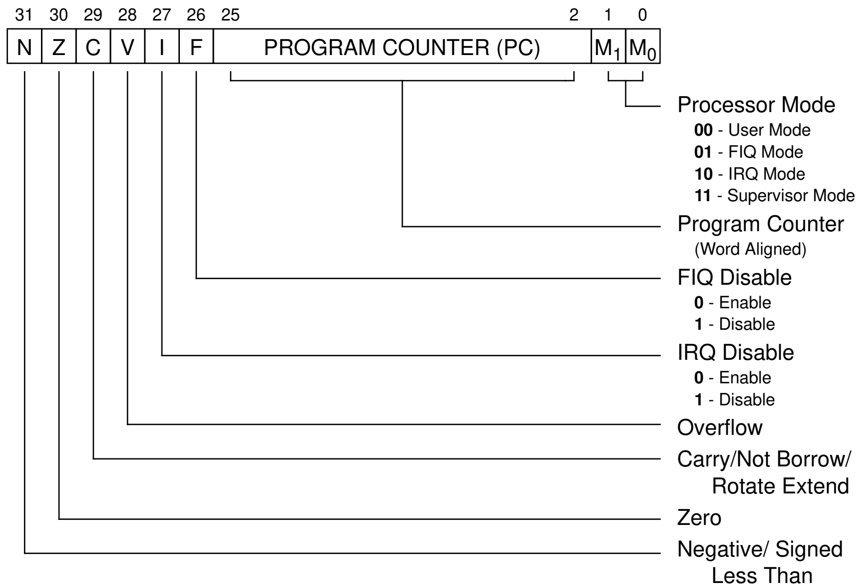


Figure A.2: The ARM Program Counter and Program Status Word

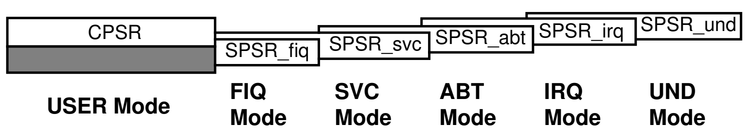


Figure A.3: The ARM6 Program Status Registers

are performed on word quantities. An unusual feature of the ARM processor, is that the entire instruction set is conditionally executable; the execution of an instruction may or may not take place depending on the value of the CPSR.

The instruction formats used by ARM6 are illustrated in figure A.4. Seven classes of instructions may be distinguished.

The first class contains data processing instructions. These perform arithmetic or logical operations on one or two operands. The first operand is always a register (Rn), while the other operand may be a shifted register or a rotated 8-bit immediate value, depending on the value of the I bit in the instruction word. The status flags of the processor may be updated according to the result of the operation if the S bit has been set. Possible results are written on a destination register (Rd),

Cond	00	I	Opcd	S	Rn	Rd	Operand 2				**Data Processing**
Cond	000000	A	S	Rd	Rn	Rs	1001	Rm			**Multiply**
Cond	0001	xx x x x xx x x x x	1xx1	xxx							**Undefined**
Cond	01	I	P	U	B	W	L	Rn	Rd	Offset	**Single Data Transfer**
Cond	011	x x x x x x x x x x xx x x x x	1	x x x							**Undefined**
Cond	100	P	U	S	W	L	Rn	Register list			**Block Data Transfer**
Cond	101	L	offset								**Branch**
Cond	110	P	U	N	W	L	Rn	CRd	CP#	offset	**Co-Proc Data Transfer**
Cond	1110	CPop	CRn	CRd	CP#	CP	0	CRm			**Co-Proc Data Op**
Cond	1110	CP	L	CRn	Rd	CP#	CP	1	CRm		**Co-Proc Register Transfer**
Cond	1111	ignored by ARM									**Software Interrupt**

Figure A.4: ARM Instruction Formats

a certain set of instructions do not produce any result; there are **CMP** (compare), **CMN** (compare negated), **TST** (bit test, equivalent to AND) and **TEQ** (test equal, equivalent to EOR). The remaining data processing instructions, which do produce a result, include the logical operations **AND**, **EOR**, **ORR**, **BIC** (bit clear), **MOV** (move), **MVN** (move not), and the arithmetic operations **ADD**, **ADC** (add with carry), **SUB**, **SBC** (subtract with carry), **RSB** (reverse subtract), **RSC** (reverse subtract with carry), **MUL** (multiply, result is the least significant 32 bits of a 32x32 product) and **MLA** (multiply accumulate, as MUL with the result initialized). All the ARM data operations (except the multiplies) allow an arbitrary shift to be applied on one of the operands before the operation is performed, so there are no separate shift instructions.

The second class of instructions are used to load (**LDR**) or store (**STR**) byte and word data values from and to memory. The memory address used in the transfer is calculated by adding to (if U bit is set) or subtracting from a base register (Rn) an offset, which may be either a register (optionally shifted) or a 12-bit immediate value. The offset may be modified either before (pre-indexed, P bit set) or after (post-index) the base is used as a transfer address. The instruction also supports optional auto increment/decrement of the base register, depending on the value of the W bit.

The instructions of the third class specify transfers of blocks of data between memory and any subset of the currently visible registers; **LDM** specifies a multiple load and **STM** a multiple store operation. The registers to be transferred are specified by the programmer by means of a 16-bit register list field, with each bit corresponding to a register. As in the single data transfer operation, the transfer addresses are determined by the base register (Rn) and the pre/post (P) and increment/decrement (U) bits. The registers are transferred in a sequential manner, with the highest in the list transferred last (R15, if specified, will always be transferred last). Multiple register transfer instructions are intented to provide an efficient mechanism for saving or restoring context, or for moving large blocks of data around main memory.

The fourth class includes instructions which specify branches. The control transfer address is calculated by shifting an offset left by two bits, and adding the result to the program counter, any overflow being ignored; the offset is a signed 2's complement 24-bit number. Thus, the branch can reach an address within +/- 32 Mbytes. Two branch variants are provided, namely simple branch (**B**) and branch with link (**BL**); the type of branch depends on the value of the L bit in the instruction word. Branch with link writes the old PC into the link register (R14) in the register bank, for subsequent use as a subroutine return address. The PC value written into R14 is adjusted to allow for the prefetch, and contains the address of the instruction following the branch and link instruction. The CPSR is not saved with the PC. To return from a subroutine the contents of R14 are restored as the new PC value (i.e. by performing a move R14 to R15 operation).

A fifth instruction class contains the software interrupt instruction (**SWI**) which is used to enter supervisor mode in a controlled manner. The instruction causes the software interrupt trap to be taken, an operation that effects the mode change. The PC is forced to a fixed value (&08) and the CPSR is saved in $SPSR_{svc}$.

The sixth class includes the **MSR** and **MRS** instructions which provide access to the CPSR and SPSR registers. MRS allows the contents of the CPSR or SPSR_mode to be moved to a general register, while MSR allows the contents of a general register to be moved to the CPSR or SPSR_mode register.

Finally, the instructions of the last class are used to support external coprocessors.

Appendix B

Modelling the Control Logic of AMULET1

This appendix illustrates the internal organization of the occarm processes that model the control circuitry of AMULET1. Emphasis is given to the use of the SEQ and PAR occam constructors in order to describe the partial orderings of events specified by the circuit. Five major processes have been selected to be presented, namely Dec1CtrlB (primary decode, see section 6.6.2), AddC (address interface, see section 6.3.2), Ctrl2 (execution unit, see section 6.8.2), Ctrl3 (execution unit, see section 6.8.3), and WrtCtrl (write bus control unit, see section 6.9).

The circuit schematics presented in the appendix have been produced within the Compass Design Automation environment and have been kindly provided by Paul Day.

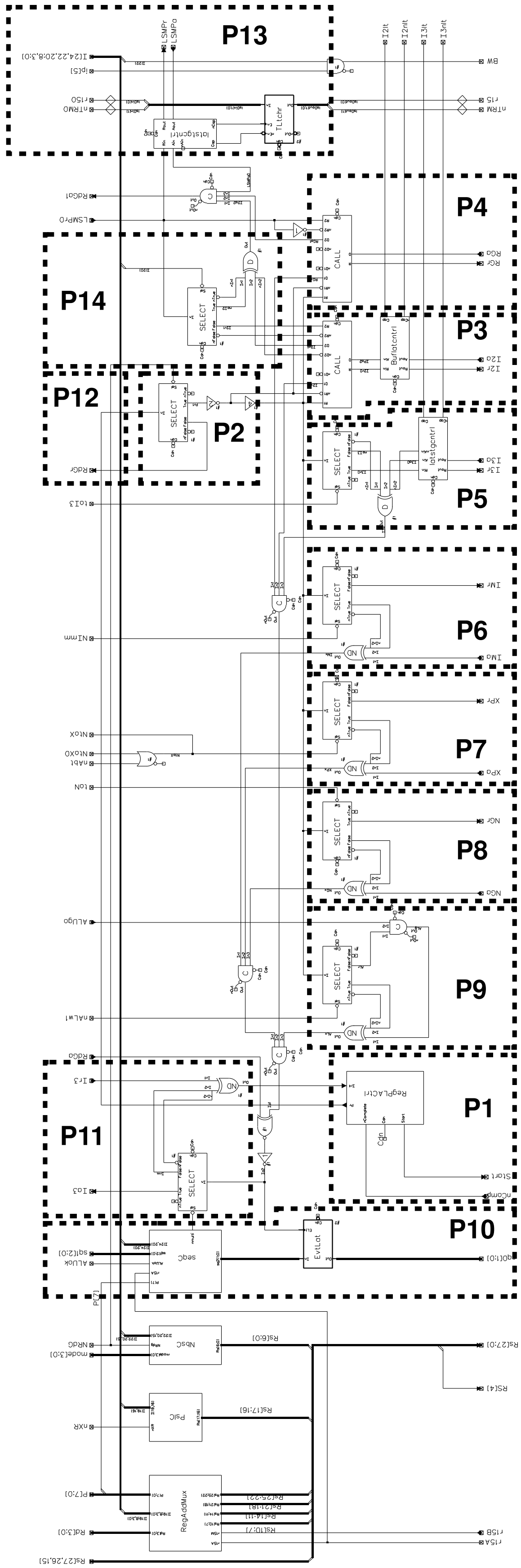


Figure B.1: Dec1CtrlB Control Circuit

```
SEQ
  P1
  P2 --IF
    TRUE
      SEQ
        PAR
          P3
          P4
          P5
          P6
          P7
          P8
          P9
    FALSE  --LDM/STM
      SEQ
        WHILE done=FALSE --interaction with RdNeg
          SEQ
            P12
            ALT  --deterministic
              LSMPr?
                SEQ
                  PAR
                    P4
                    P14
                    P13
              RdGA?
                SEQ
                  done:=TRUE
  P10
  P11
:
```

Figure B.2: The Dec1CtrlB Process

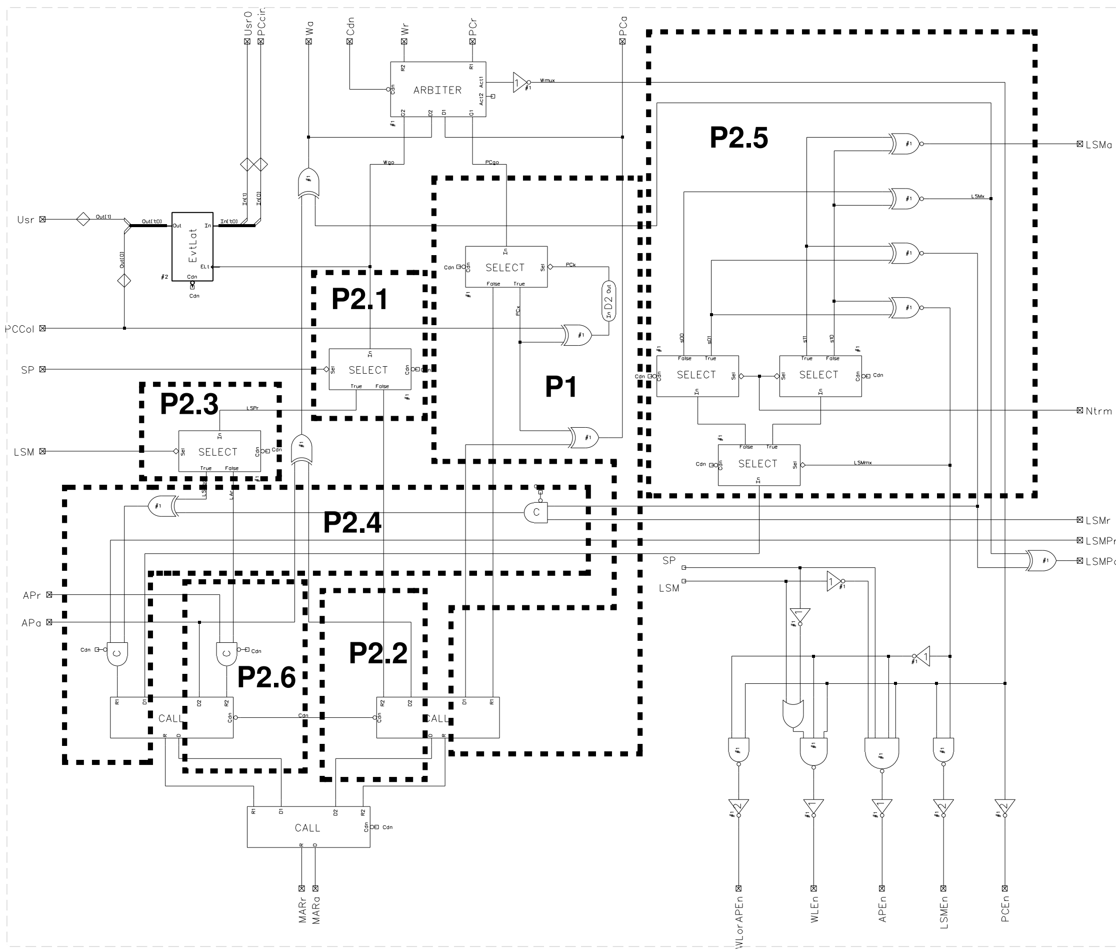


Figure B.3: AddC Control Circuit

```
SEQ
  ALT
    PCr?
      P1            --PC loop
    Wr?
      P2.1          --IF
        FALSE
          SEQ
            P2.2   --data transfer
        TRUE
          SEQ
            P2.3  --IF
              TRUE
                SEQ
                  WHILE Ntrm=FALSE  -- LSM loop
                    SEQ
                      P2.4
                      P2.5
              FALSE
                SEQ
                  P2.6 --data transfer: Apipe
:
```

Figure B.4: The AddC Process

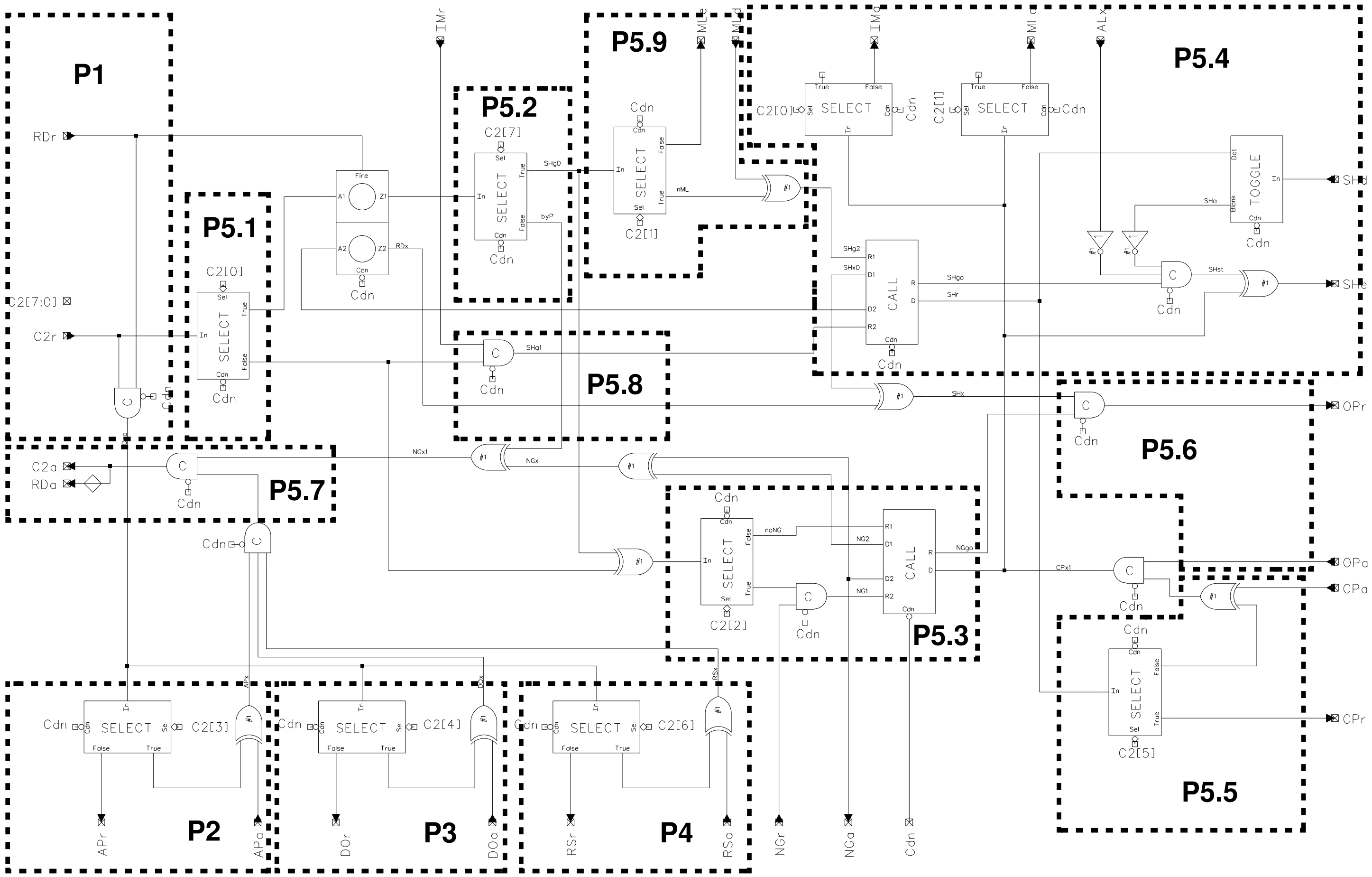


Figure B.5: Ctrl2 Control Circuit

```
SEQ
  P1
  PAR
    P2
    P3
    P4
    SEQ --P5
      P5.1 --IF
        TRUE
          P5.2 --IF
            TRUE
              SEQ
                PAR
                  P5.3
                  SEQ
                    P5.9  --Multiplier
                    P5.4  --Shifter
                PAR
                  P5.5
                  P5.6
            FALSE
              SKIP
        FALSE
          SEQ
            PAR
              P5.3
              SEQ
                P5.8
                P5.4  --Shifter
            PAR
              P5.5
              P5.6
  P5.7
:
```

Figure B.6: The Ctrl2 process

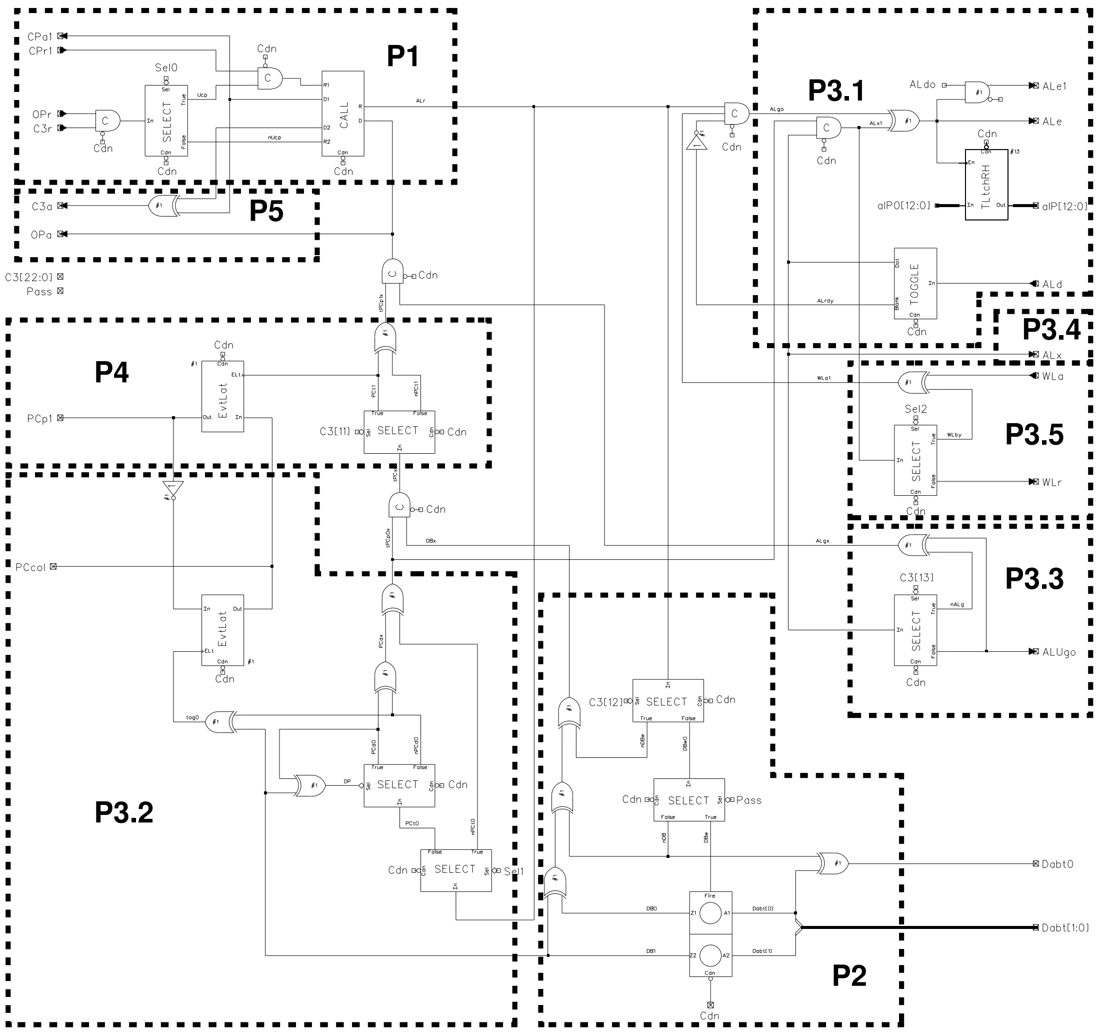


Figure B.7: Ctrl3 Control Circuit

```
SEQ
  P1 --input and CPSR
  PAR
    P2
    SEQ --P3
      P3.1
      PAR
        P3.4
        P3.3
        SEQ
          P3.2
          P3.5
  P4
  P5
:
```

Figure B.8: The Ctrl3 process

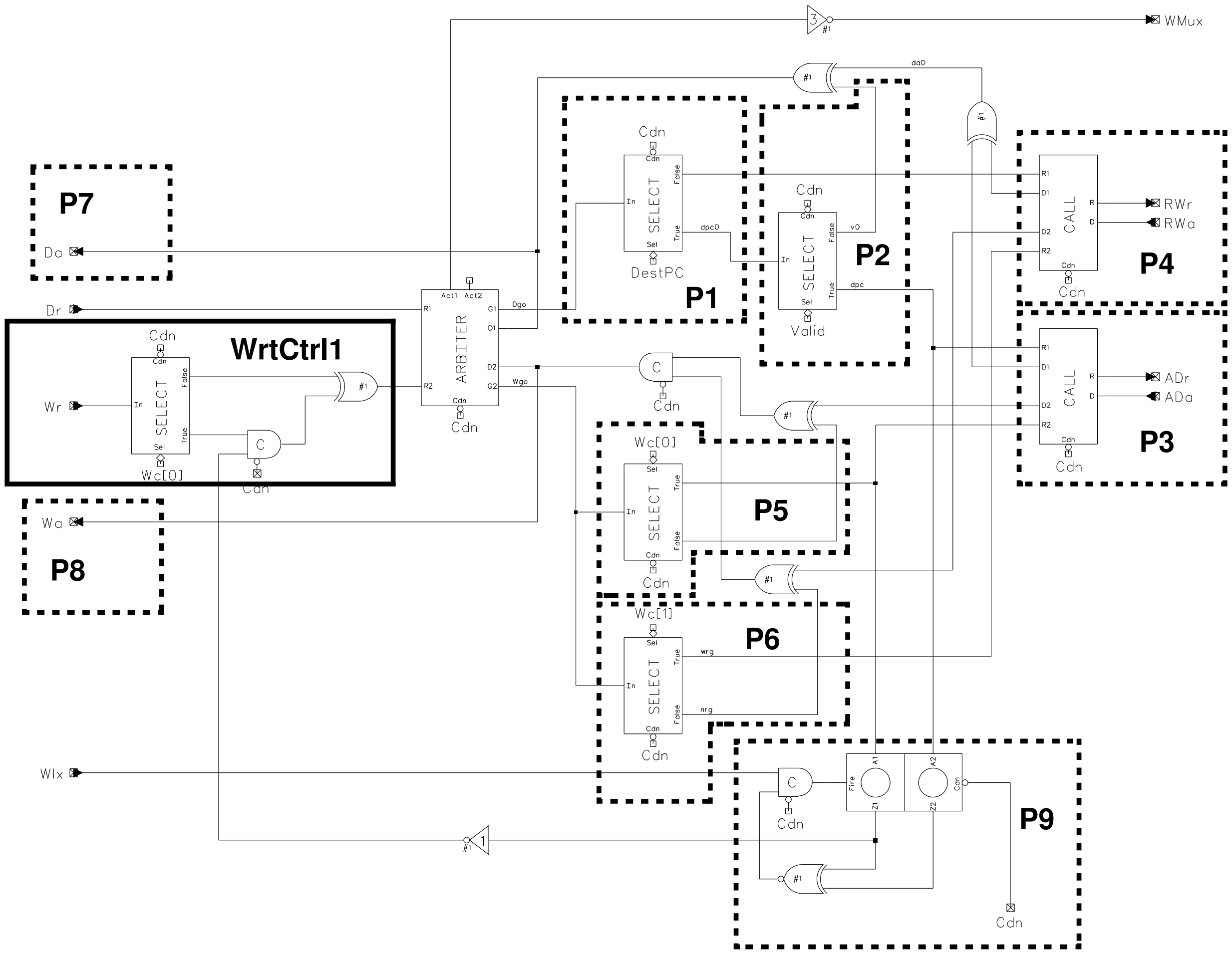


Figure B.9: Write Control Circuit

```
ALT
  Dr? --from data interface
    SEQ
      P1 --IF
        TRUE
          SEQ
            P2 --IF
              TRUE
                PAR
                  P3
                  P9
              FALSE
                SKIP
        FALSE
          SEQ
            P4
      P7
  Wr? --from execution unit
    SEQ
      PAR
        SEQ
          P5 --IF
            TRUE
              PAR
                P3
                P9
            FALSE
              SKIP
        SEQ
          P6 --IF
            TRUE
              SEQ
                P4
            FALSE
              SKIP
      P8
:
```

Figure B.10: The WrtCtrl2 process

Bibliography

[AMD87] "Am29000 User's Manual", Advanced Micro Devices, 1987.

[AMUL] "The AMULET Group", World Wide Web Home Page, URL: http://www.cs.man.ac.uk/amulet/index.html

[OMI94] "Report on the Integer Macrocell Revision", ESPRIT Project 6909 - OMI/DE - ARM, Deliverable 7.4.2, University of Manchester, May 1994.

[Agra87] Agrawal, P., et al., "MARS: A Multiprocessor Based Multiprogrammable Accelerator", IEEE Design and Test of Computers, 4, 10, October 1987, pp. 28-36.

[Ahuj86] Ahuja, S., Carriero, N., Gelernter, D., "Linda and Friends", IEEE Computer, August 1986, pp. 26-34.

[Akel91] Akella, V., Gopalakrishnan, G., "Hopcp: A Concurrent Hardware Description Language", Technical Report UUCS-TR-91-021, Department of Computer Science, University of Utah, 1991.

[Alli92] Allison, A., "Is RISC Really Doomed?", Microprocessor Report, Issue 47, October 1992.

[Alma89] Almasi, G. S., Gottlieb, A. J., "Highly Parallel Computing ". The Benjamin/Cummings Publishing Company Inc., 1989.

[Alme94] Almeida, F. A., Welch, P. H., "A Parallel Emulator for a Multiprocessor Dataflow Machine", Proceedings of the World Transputer Congress 1994, Como, September 1994, pp. 259-272.

[Alon93] Alonso, J. M., et al., "Conservative Parallel Discrete Event Simulation in a Transputer-Based Multicomputer", Proceedings of the World Transputer Congress 1993, Aachen, September 1993, pp. 636-650.

[Aris] Aristotle, "Physics", English Translation: Loeb Classical Library, Harvard University Press.

[Aris-1] Aristotle, "Metaphysics", English Translation: Loeb Classical Library, Harvard University Press.

[Arms89] Armstrong, J. R., "Chip Level Modelling with VHDL", Prentice Hall International, 1989.

[Akyi93] Akyildiz, I. F., et al., "The Effect of Memory Capacity on Time Warp Performance", Journal of Parallel and Distributed Computing, 18, 4, August 1993, pp. 411-422.

[Amda67] Amdahl, G. M., "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities", Proceedings AFIPS 1967 Sping Joint Computer Conference, Atlantic City, April 1967, pp. 483-485.

[Asch77] Aschcroft, E. A., Wadge, W. W., "LUCID: A Nonprocedural Language with Iteration", Communications of the ACM, 20, 7, July 1977, pp. 519-526.

[Aust79] Austin, J.H., "The Burroughs Scientific Processor", Infotech State of the Art Report: Supercomputers, Vol. 2, 1979, pp. 1-31.

[Avra83] Avramovici, M., Levendel, Y. M., Memon, P. R., "A Logic Simulation Engine", IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, 2, 2, April 1983.

[Ayan89] Ayani, R., "A Parallel Simulation Scheme based on the Distance Between Objects", Proceedings of the 1989 SCS Multiconference on Distributed Simulation, SCS Simulation Series, March 1989, pp. 113-118.

[Ayan90] Ayani, R., Rajaei, H., "Parallel Simulation of a Generalized Cube Multistage Interconnection Networks", Proceedings of the 1990 SCS Multiconference on Distributed Simulation, SCS Simulation Series, January 1990, pp. 60-63.

[Ayan92] Ayani, R., Rajaei, H., "Parallel Simulation using Conservative Time Windows", Proceedings of the 1992 Winter Simulation Conference, December 1992, pp. 709-717.

[ARM] ARM Ltd, Fulbourn Road, Cherry Hinton, Cambridge, CB1 4JN, England.

[Baba93] Babaoglou, O., Marzullo, K., "Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms", Technical Report UBLCS-93-1, Laboratory for Computer Science, University of Bologna, January 1993.

[Bacc93] Baccelli, F., Canales, M., "Parallel Simulation of Stochastic Petri Nets Using Recurrence Equation", ACM Transactions on Modeling and Computer Simulation, 3, 1, January 1993, pp. 20-41.

[Back78] Backus, J., "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs", Communications of the ACM, 21, 8, August 1978, pp. 613-641.

[Bagr90] Bagrodia, R. L., Liao, W. T., "Parallel Simulation of the Sharks World Problem", Proceedings of the 1990 Winter Simulation Conference, December 1990, pp. 191-198.

[Baik85] Baik, D., Zeigler, B. P., "Performance Evaluation of Hierarchical Distributed Simulators", Proceedings of the 1985 Winter Simulation Conference, December 1985, pp. 421-427.

[Bail91] Bailey, "Measuring the Overhead in Conservative Parallel Simulations of Multicomputer Programs", Proceedings of the 1990 Winter Simulation Conference, December 1990, pp. 627-636.

[Bain88] Bain, W. L., Scott, D. S., "An Algorithm for Time Synchronization in Distributed Discrete Event Simulation", Proceedings of the 1988 SCS Multiconference on Distributed Simulation, SCS Simulation Series, July 1988, pp. 30-33.

[Balc94] Balci, O., "Validation, Verification and Testing Techniques Throughout the Life Cycle of a Simulation Study", Proceedings of the 1994 Winter Simulation Conference", December 1994, pp. 215-220.

[Balc94a] Balci, O., "Principles of Simulation Model Validation Verification and Testing" Technical Report TR-94-24, Departmnet of Computer Science, Virginia Tech, Virginia, USA, 1994.

[Balm90] Balmer, D. W., Paul, R. J., "Integrated Support Environments for Simulation Modelling" Proceedings of the 1990 Winter Simulation Conference, December 1990, pp. 243-249.

[Bank86] Banks, J., Carson, J. S., "Introduction to Discrete Event Simulation", Proceedings of the 1986 Winter Simulation Conference, December 1986, pp. 17-23.

[Bank92] Banks, J., "Simulation Languages and Simulator Environments", Proceedings of the 1992 Winter Simulation Conference, December 1992, pp. 88-96.

[Barn89] Barnes, J. G. P., "Programming in Ada", Addison Wesley, 1989.

[Bart89] Barton, R. R, Schruben, L. W., "Graphical Methods for the Design and Analysis of Simulation Experiments", Proceedings of the 1989 Winter Simulation Conference, December 1989, pp. 51-61.

[Bast94] Basten T., et al. "Time and the Order of Abstract Events in Distributed Computations", Technical Report TR-94-01, University of Waterloo, Canada, February 1994.

[Batc80] Batcher, K.E., "The Design of a Massively Parallel Processor", IEEE Transactions on Computers,29, 9, September 1980, pp. 836-840.

[Bell90] Bellenot, S., "Global Virtual Time Algorithms", Proceedings of the 1990 SCS Multiconference on Distributed Simulation, SCS Simulation Series, January 1990, pp. 122-127.

[Bell92] Bellenot, S., "State Skipping Performance with the Time Warp Operating System", Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS92), SCS, January 1992, pp. 53-64.

[Bemm90] Bemmerl, T., et al., "TOPSYS: Tools for Parallel Systems", Technical Report TUM-I9047, SFB-Bericht Nr. 342/25/90 A, Institute fur Informatik der Technischen Universitat Munchen, January 1990.

[Berr85] Berry, O., Jefferson , D., "Critical Path Analysis of Distributed Simulation", Proceedings of the SCS Distributed Simulation Conference, SCS Simulation Series, 1985, pp. 57-60.

[Bile85] Biles, W. E., Daniels, D. M., O'Donnel, T. J., "Statistical Considerations in Simulation on a Network of Microcomputers", Proceedings of the 1985 Winter Simulation Conference, December 1985, pp. 388-393.

[Birt94] Birtwistle, G., Private Communication, 1994.

[Birt94a] Birtwistle, G., Liu, Y., "Specification of the Manchester AMULET1: Top Level Specification", Technical Report, Computer Science Department, University of Calgary, December 1994.

[Birt95] "Asynchronous Digital Circuit Design", Editors Birtwistle, G., Davis, A, Springer Verlang, 1995.

[Blan84] Blank, T., "A Survey of Hardware Accelerators Used in Computer Aided Design", IEEE Design and Test of Computers, 1, 8, August 1984, pp. 21-39.

[Böhm91] Böhm, A. P. W., et al., "SISAL 2.0 Reference Manual", Technical Report Cs-91-118, Department of Computer Science, Colorado State University, 1991.

[Boil87] Boillat, J. E., et al., "An Analysis and Reconfiguration Tool for Mapping Parallel Programs onto Transputer Networks", Proceedings of the 7th Occam Users group, September 1987, pp. 186-194.

[Bott86] Bottomley, R., "The Meiko Computing Surface: A Configurable Supercomputer", IEE Colloquium: The Transputer: Applications and Case Studies, IEE Digest, 1986/91, 23rd May 1986.

[Bouk94] Boukerche, A., Tropper, C., "A Static Partitioning and Mapping Algorithm for Conservative Parallel Simulations", Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS94), SCS, July 1994, pp. 164-172.

[Brin91] Briner, J. Jr., "Fast Parallel Simulation of Digital Systems", SCS Advances in Parallel and Distributed Simulation, 23, January 1991, pp. 71-77.

[Broz89] Brozowski, J. A., Ebergen, J. C., "Recent Developments in the Design of Asynchronous Circuits", Research Report CS-89-18, Computer Science Department, University of Waterllo, May 1989.

[Broz95] Brozowski, J. A., Seger, C-J., H., "Asynchronous Circuits", Springer Verlang, 1995.

[Brun89] Brunvand, E., Sproull, R. F., "Translating Concurrent Programs into Delay-Insensitive Circuits", Proceedings of ICCAD, 1989, pp. 262-265.

[Brun91] Brunvand, E., "Translating Concurrent Communicating Programs into Asynchronous Circuits", Ph.D Thesis, Carnegie Mellon University, 1991.

[Brun91a] Brunvand, E., Starkey, M., "An Integrated Environment for the Design and Simulation of Self Timed Systems", Proceedings of VLSI 1991, August 1991. pp. 4a.2.1-4a.3.1.

[Brun93] Brunvand, E., "The NSR Processor", Proceedings of the 26th Hawaii International Conference on System Sciences (HICSS 1993), January 1993, pp. 428-435.

[Brya77] Bryant, R. E., "Simulation of Packet Communication Architecture Computer Systems", Technical Report MIT-LCS-TR-188, Laboratory for Computer Science, Massachusetts Institute of Technology, USA, 1977.

[Brze93] Brzezinski, J., Helary, J. M., Raynal, M., "Distributed Termination Detection: General Model and Algorithms", Technical Report 1964, INRIA, March 1993

[Bund86] Bunday, B. D., "Basic Queueing Theory", Edward Arnold, 1986.

[Burn87] Burns, S. M., Martin, A. J., "Synthesis of Self-Timed Circuits by Program Transformations", Technical Report 5253:TR:87, Computer Science Department, Caltech,1987.

[Burn88] Burns, A., "Programming in Occam 2", Addison Wesley, 1988.

[Cai90] Cai, W., Turner, S. J., "An Algorithm for Discrete Event Simulation: The Carrier Null Message Approach" Proceedings of the 1990 SCS Multiconference on Distributed Simulation, SCS Simulation Series, January 1990, pp. 3-8.

[Cai90a] Cai, W., Turner, S. J., "Experimental Studies of Conservative Distributed Discrete Event Simulation on Transputer Networks", Proceedings of the 12th Occam User Group Technical Meeting, IOS Press, 1990, pp. 138-147.

[Capo86] Capon, P.C., Gurd, J. R., Knowles, A. E., "ParSiFal: A Parallel Simulation Facility", IEE Colloquium: The Transputer: Applications and Case Studies, IEE Digest, 1986/91, 23rd May 1986.

[Cars89] Carson, J. S., "Verification and Validation: A Consultan's Perspective", Proceedings of the 1989 Winter Simulation Conference, December 1989, pp. 552-558.

[Cars92] Carson, J. S., "Modelling". Proceedings of the 1992 Winter Simulation Conference, December 1992, pp. 82-87.

[Chan79] Chandy, K. M., Holmes, V., Misra, J., "Distributed Simulation of Networks", Computer Networks, 3, 1, January 1979, pp. 105-113.

[Chan79a] Chandy, K. M., Misra, J., "Distributed Simulation: A Case Study in the Design and Verification of Distributed Programs", IEEE Transactions on Software Engineering, 5, 5, September 1979, pp. 440-452.

[Chan79b] Chandy, K. M., Misra, J., "Deadlock Absence Proofs for Networks of Communicating Processes", Information Processing Letters, 9, 4, November 1979, pp. 185-189.

[Chan81] Chandy, K. M., Misra, J., "Asynchronous Distributed Simulation via a Sequence of Parallel Computations", Communications of the ACM, 24, 11, November 1981, pp. 198-205.

[Chan85] Chandy, K. M., Lamport, L., "Distributed Snapshots: Determining Global States of Distributed Systems", ACM Transactions on Computer Systems, 3, 1, February 1985, pp. 63-75.

[Chan89] Chandy, K. M., Sherman, R., "The Conditional Event Approach to Distributed Simulation", Proceedings of the 1989 SCS Multiconference on Distributed Simulation, SCS Simulation Series, March 1989, pp. 93-99.

[Chiu94] Chiu, P. P. K., Ku, K. M., "Simulation of Logic Circuits Using a Network of Transputers", Proceedings of the World Transputer Congress 1994, Como, September 1994, pp. 273-284.

[Chri89] Christian, F., "probabilistic Clock Synchronization", Distributed Computing, 3, 3, March 1989, pp. 146-158.

[Chri82] Christopher, T., et al., "Structure of a Distributed Simulation System", IEEE COMPSAC 1982, IEEE Computer Society Press, pp. 548-589.

[Chu85] Chu, T. A., Leung, C. K. C., Wanuga, T. S., "A Design Methodology for Concurrent VLSI systems", Proceedings of ICCD 1985, 1985, pp. 407-410.

[Chu86] Chu, T. A., "On the Models for Designing VLSI Asynchronous Digital Systems", INTEGRATION, the VLSI Journal, 4, 1986. pp. 99-113,

[Chu86a] Chu, T. A., Glasser, L. A., "Synthesis of Self-timed Control Circuits from Graphs : An Example", Proceedings of ICCD 1986, 1986, pp.565-571.

[Chu87] Chu, T. A., "Synthesis of Self-timed VLSI Circuits from Graph-Theoretic Specifications", Ph.D Thesis (MIT/LCS/TR-393), M.I.T., June 1987.

[Chun91] Chung, M., Chung, Y., "An Experimental Analysis of Simulation Clock Advancement in Parallel Logic Simulation on an SIMD Machine", SCS Advances in Parallel and Distributed Simulation, 23, January 1991, pp. 125-132.

[Clar67] Clark, W. A., "Macromodular Computer Systems", Proceedings of the 1967 AFIPS Spring Joint Computing Conference, April 1967 pp. 335-336.

[Clar74] Clark, W. A., Molnar C. E., "Macromodular Computer Systems. Computers", Chapter 3, In Biomedical Research, Editors Stacy R. W., Waxman B.D, Academic Press, 1974.

[Clea94] Cleary, J., et al., "Cost of State Saving and Rollback", Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS94), SCS, July 1994, pp. 94-101.

[Cloc81] Clocksin, W. F., Mellish, C. S., "Programming in Prolog", Springer Verlag, 1981.

[Coat93] Coates, B., Davis, A., Stevens, K. S., "Automatic Synthesis of Fast Compact Self-timed Control Circuits", Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies, Manchester, England, 1993.

[Comf84] Comfort, J. C., "The Simulation of a Master-Slave Event Set Processor", Simulation, 42, 3, March 1984, pp. 117-124.

[Comf88] Comfort, J. C., Gopal, P. R., "Environment Partitioned Distributed Simulation with Transputers", Proceedings of the 1988 SCS Multiconference on Distributed Simulation, SCS Simulation Series, July 1988, pp. 103-108.

[Conc89] Conception, A. I., "A hierarchical Computer Architecture for Distributed Simulation", IEEE Transactions on Computers 38, 2, February 89, pp. 311-319.

[Cota90] Cota, B. A., Sargent, R. G., "A Framework for Automatic Lookahead Computation in Conservative Distributed Simulations", Proceedings of the 1990 SCS Multiconference on Distributed Simulation, SCS Simulation Series, January 1990, pp. 56-59.

[Davi89] David, I., Ginosar, R., Yoeli, M., "Self-Timed Implementation of a Reduced Instruction Set Computer", Technical Report 732, Technion and Israel Institute of Technology, October 1989.

[Davi88] Davis, C. K., Sheppard, S. V., Lively, W. M., "Automatic Development of Parallel Simulation Models in Ada", Proceedings of the 1988 Simulation Conference, December 1988, pp. 339-343.

[Davi95] Davis, A., Nowick, S. M., "Asynchronous Circuit Design: Motivation, Background and Methods", In [Birt95], pp. 1-49.

[Davi95a] Davis, A., Nowick, S. M., "Synthesizing Asynchronous Circuits: Practice and Experience", In [Birt95], pp. 104-150.

[Day92] Day, P., "A Micropipelined Multiplier", Proceedings of the ACiD-WG/EXACT Workshop on Asynchronous Data Processing, Veldhoven, The Netherlands, December 1992.

[Day95] Day, P., "Investigations into Micropipeline Latch Design Styles", to be published in IEEE Transactions in VLSI.

[DeBe88] DeBenectitus, E. P., Ackland, B. D., "Circuit Simulation on a Hypercube", Proceedings of the 1988 SCS Multiconference on Distributed Simulation, SCS Simulation Series, July 1988, pp. 89-93.

[DeBe91] DeBenectitus, E. P, Ghosh, S., Yu. M. L., "A Novel Algorithm for Discrete Event Simulation", Computer, 24, 6, June 1991, pp. 21-33.

[DeRe76] DeRemer, F., Kron, H. H., "Programming-in-the-Large versus Programming-in-the-Small", IEEE Transactions on Software Engineering, 2, 2, February 1976, pp. 114-121.

[Dean92] Dean, M. E., "STRiP: A Self-Timed RISC Processor Architecture", Ph.D Thesis, Stanford University, 1992.

[Derr89] Derrick, E. J., Balci, O., Nance, R. E., "A Comparison of Selected Conceptual Frameworks for Simulation Modelling", Proceedings of the 1989 Winter Simulation Conference, December 1989, pp. 711-718.

[DeVr90] DeVries, R. C., "Reducing Null Messages in Misras Distributed Discrete Event Simulation Method", IEEE Transactions on Software Engineering, 16, 1, January 1990, pp. 82-91.

[Dick90] Dickens, P. M., Reynolds, P. F., "SRADS With Local Rollback", Proceedings of the 1990 SCS Multiconference on Distributed Simulation, SCS Simulation Series, January 1990, pp. 161-164.

[Dijk68] Dijkstra, E.W., "The Structure of THE Multiprogramming System", Communications of the ACM, 11, 5, May 1968, pp. 341-346.

[Dijk75] Dijkstra, E.W., "Guarded Commands, Nondeterminacy and Formal Derivation of Programs", Communications of the ACM, 18, 8, August 1975, pp. 453-457.

[Dijk80] Dijkstra, E. W., Scholten, C. S., "Termination Detection for Diffusing Computations", Information Processing Letters, 11, 1, August, 1980.

[Dill89] Dill, D. L., "ACM Distinguished Dissertations: Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits", MIT Press, 1989.

[Ditz87] Ditzel, D. R., McLellan, H. R., Berenbaum, A. D., "The Hardware Architecture of the CRISP Microprocessor", Proceedings of the 14th Annual Symposium on Computer Architecture, 1987, pp. 309-319.

[Djan89] Djanhaguir, A. H., Geffroy, J. C., "Use of Occam for Validation of Distributed Event Driven Simulation", Proceedings of the 10th Occam User Group Technical Meeting, Enschede, April 1989, pp. 213-221.

[Dobb92] Dobberpuhl, D., et al., "A 200 MHz 64b Dual-Issue CMOS Microprocessor", IEEE Journal of Solid-State Circuits, 27, 11, November 1992, pp. 155-1565.

[Dobb88] Dobbs, C., Reed, P., Ng, T., "Supercomputing on a Chip: Genesis of the 88000", VLSI System Design, May 1988, pp. 24-33.

[Dows85] Dowsing, R. D., "Simulating Hardware Structures in occam", Software and Microsystems, 4, 4, August 1985, pp. 77-84.

[Dows88] Dowsing, R. D., "Introduction to Concurrency Using Occam", Van Nostrand Reinhold, 1988.

[Duga35] Dugas, R., "A History of Mechanics", English Translation of the Original 1935 French Edition, Central Book Company, New York, 1955.

[Dunc90] Duncan, R., "A survey of Parallel Computer Architectures", Computer, February 1990, pp. 5-16.

[Duni91] Dunigan, T. H., "Hypercube Clock Synchronization", Technical Report TM-11744, Oak Ridge National Laboratory, February 1991.

[Dynk82] Dynkin, E. B., "Markov Processes and Related Problems of Analysis", Cambridge University Press, 1982.

[Eber91] Ebergen, J. C., "A Formal Approach to Designing Delay-Insensitive Circuits", Distributed Computing, 5, 3, March 1991, pp. 107-119.

[Ebli89] Ebling, M., et al., "An Ant Foraging Model Implemented on the Time Warp Operating System", Proceedings of the 1989 SCS Multiconference on Distributed Simulation, SCS Simulation Series, March 1989, pp. 151-163.

[Eick91] Eick, S. G., et al., "Synchronous Relaxation for Parallel Simulations with Applications to Circuit-Switched Networks", Proceedings of the 5th Workshop on Parallel and Distributed Simulation (PADS91), SCS, January 1991, pp. 151-163.

[Eshr89] Eshraghian, K., Weste, N., "Principles of CMOS Design A Systems Perspective", Addison Wesley, 1989.

[FDR93] FDR
User Manual (available via anonymous ftp at ftp.comlab.ox.ac.uk), Formal Systems Europe, 3 Alfred Street, Oxford, 1993.

[Feld79] Feldman, J. A., "High Level Programming for Distributed Computing", Communications of the ACM, 22, 6, June 1979, pp. 353-368.

[Fers93] Ferscha, A., Chiola, G., "Distributed Simulation of Timed Petri Nets: Exploiting the Net Structure to Obtain Efficiency", Proceedings of the 14th International Conference on Application an and Theory of Petri Nets, Lecture Notes in Computer Science, 691, 1993, pp. 146-165.

[Fers94] Ferscha, A., Tripathi, S. K., "Parallel and Distributed Simulation if Discrete Event Systems", Technical Report CS.TR.3336, University of Maryland, August 1994.

[Fidg88] Fidge, C. J., "Timestamps in Message Passing Systems that Preserve the Partial Ordering", In Proceedings of the 11th Australian Computer Science Conference, Brisbane, Australia, February 1988, pp. 55-66.

[Fidg91] Fidge, C. J., "Logical Time in Distributed Computing Systems", IEEE Computer, 24, 8, August 1991, pp. 28-33.

[Fish92] Fishwick, P. A., "Integrating Modeling in Simulation, Software Engineering and AI", in [Radi92], pp. 773-774.

[Flec92] Fleckenstein, C. J., et al., "Multiprocessing", in [Yovi92], pp. 255-324.

[Flyn72] Flynn, M., "Some Computer Organizations and their Effectiveness", IEEE Transactions on Computers, 21, 9, September 1972, pp. 948-960.

[Fox89] Fox, G. C., "Parallel Computing Comes of Age: Supercomputer Level Parallel Computations at Caltech", Concurrency: Practice and Experience, 1, 1, John Wiley, September 1989, pp. 63-104.

[Franc80] Francez, N., "Distributed Termination", ACM TOPLAS, 2, 1, 1980, pp. 42-55.

[Fuji88] Fujimoto R., "Applications: Distributed Simulation", in [Reed88], Chapter 6, pp. 240-267.

[Fuji88a] Fujimoto R., Tsai, J., Gopalakrishnan, G., "Design and Performance of Special Purpose Hardware for Time Warp", Proceedings of the 15th Annual Symposium on Computer Architecture, June 1988, pp. 401-408.

[Fuji89] Fujimoto R., "Performance measurements of Distributed Simulation Strategies", Transactions of the Society for Computer Simulation, 6, 2, April 1989, pp. 88-132.

[Fuji89a] Fujimoto R., "Time Warp on a Shared Memory Multiprocessor", Transactions of the Society for Computer Simulation, 6, 3, July 1989, pp. 211-239.

[Fuji89b] Fujimoto R., "The Virtual Time Machine", Proceedings of the 1989 International Symposium on Parallel Algorithms and Architectures", June 1989, pp. 199-208.

[Fuji90] Fujimoto R., "Parallel Discrete Event Simulation", Communications of the ACM, 33, 10, October 1990, pp. 31-53.

[Fuji92] Fujimoto R., Nicol, D., "State of the Art in Computer Simulation", Proceedings of the 1992 Winter Simulation Conference, December 1992, pp. 246-254.

[Fuji93] Fujimoto R., "Parallel and Distributed Discrete Event Simulation: Algorithms and Applications", Proceedings of the 1993 Winter Simulation Conference, December 1993, pp. 106-114.

[Fuji87] "MB86900 High Performance 32-bit RISC SPARC Data Sheet", Fujitsu Microelectronics Inc., 1987.

[Furb89] Furber, S. B., "VLSI RISC Architecture and Organization", Marcel Dekker, Inc., 1989.

[Furb92] Furber, S. B., "Micropipelines - A Case Study", Proceedings of the ACiD-WG/EXACT Workshop on Asynchronous Data Processing, Veldhoven, The Netherlands, December 1992.

[Furb93] Furber, S. B., "AMULET1 - An Asynchronous ARM Processor", Symposium Record of Hot Chips V, Stanford University, August 1993.

[Furb93a] Furber, S. B., et al., "A Micropipelined ARM", Proceedings of VLSI '93 (Best Paper Award), September 1993, pp. 5.4.1-5.5.8.

[Furb94] Furber, S. B., "AMULET1: A Micropipelined ARM", IEEE CompCon '94, Invited Paper, March 1994.

[Furb94a] Furber, S. B., "The Design and Evaluation of an Asynchronous Microprocessor", Proceedings of ICCD 1994, October 1994, pp. 217-220.

[Furb94b] Furber, S. B., Talk at the AMULET Modelling Workshop, Windermere, Cumbria, England, July 1994.

[Furb94c] Furber, S. B., Private Communication.

[Furb95] Furber, S. B., "Computing Without Clocks", In [Birt95], pp. 211-262.

[Gafn85] Gafni, A., "Space Management and Cancellation Mechanisms for Time Warp", Technical Report, TR-85-3-41, Department of Computer Science, University of Southern California, 1985.

[Gafn88] Gafni, A., "Rollback Mechanisms for Optimistic Distributed Simulation Systems", Proceedings of the 1988 SCS Multiconference on Distributed Simulation, SCS Simulation Series, July 1988, pp. 61-67.

[Gajs83] Gajski, D. et al., "Cedar-A large scale multiprocessor", IEEE Proceedings of the 1983 International Conference on Parallel Processing, 1983, pp. 524-529.

[Gall90] Galletly, J., "Occam2", Pitman, 1990.

[Garc84] Garcia-Molina, H., Germano, F., Hohler, W. H., "Debugging a Distributed Computing System", IEEE Transactions on Software Engineering, 10, 2, February 1984, pp. 210-219.

[Gars89] Garside J. D., "T-Rack Switch Card", ParSiFal Internal Document PSF/MU/WP5/89/3, Department of Computer Science, University of Manchester, February 1989.

[Gars92] Garside, J. D., "Micropipeline Structures", Proceedings of the ACiD-WG/EXACT Workshop on Asynchronous Data Processing, Veldhoven, The Netherlands, December 1992.

[Gars93] Garside J. D., "A CMOS VLSI Implementation of an Asynchronous ALU", Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies, Manchester, England, 1993.

[Gill62] Gill, A., "Introduction to the Theory of Finite State Machines", McGraw Hill, 1962.

[Gilm86] Gilmer, J. B., Hong, J. P., "Replicated State Space Approach for Parallel Simulation", Proceedings of the 1986 Winter Simulation Conference, December 1986, pp. 430-433.

[Gino90] Ginosar, R., Michell, N., "On the Potential of Asynchronous Pipelined Processors", Technical Report UUCS-90-015, VLSI Systems Research Group, University of Utah, 1990.

[Glas84] Glaser, H., Hankin, C., Till, D., "Principles of Functional Programming", Prentice Hall International, 1984.

[Glaz92] Glazer, D. W., "Load Balancing Parallel Discrete Event Simulations", Ph.D. Thesis, Department of Computer Science, McGill University, 1992.

[Gold88] Goldsmith, M., Jones, G., "Programming in occam 2", Prentice Hall International, 1988.

[Gopa90] Gopalakrishnan G., Jain P., "Some Recent Asynchronous System Design Methodologies", Technical Report UU-CS-TR-90-016, Department of Computer Science, University of Utah, October 1990.

[Gopa93] , Gopalakrishnan, G., Akella, V., "Specification, Simulation, and Synthesis of Self-Timed Circuits", Proceedings of the 26th Hawaii International Conference on System Sciences (HICSS 1993), January 1993. pp. 399-408.

[Gord90] Gordon, R. F., et al., "Hierarchical Modelling in a Graphical Simulation System", Proceedings of the 1990 Winter Simulation Conference, December 1990, pp. 499-503.

[Grah94] Graham, R. L., Knuth, D. E., Patashnik, O., "Concrete Mathematics", Addison Wesley, 1994.

[Gros88] Groselj, B. Tropper, C., "The Time of Next Event Algorithm", Proceedings of the 1988 SCS Multiconference on Distributed Simulation, SCS Simulation Series, July 1988, pp. 25-29.

[Gros89] Groselj, B. Tropper, C., "A Deadlock Resolution Scheme for Distributed Simulation", Proceedings of the 1989 SCS Multiconference on Distributed Simulation, SCS Simulation Series, March 1989, pp. 108-112.

[Gurd85] Gurd, J.R. et al., "The Manchester Prototype Dataflow Computer", Communications of the ACM, 28, 1, January 1985, pp. 34-52.

[Guse84] Gusella, R., Zatti, S., "Tempo: A Network Time Controller for a Distributed Berkeley Unix System", Distributed Processing Technical Communication Letter, IEEE, SI-6, June 1984, pp. 7-15.

[Gwen94] Gwennap, L., "Digital Leads the Pack with 21164", Microprocessor Report, 8, 12, September 1994, pp. 1, 6-10.

[HP86] "Precision Architecture and Instruction Reference Manual", Hewlett-Packard Company, 1986.

[Hans73] Hansen, B. P., "Operating Systems Principles", Prentice Hall International, 1973.

[Hart87] "Hardware Description Languages", Advances in CAD for VLSI, Volume 7, Hartenstein, R. W., (Editor), North Holland, 1987.

[Hauc93] Hauck, S., "Asynchronous Design Methodologies: An Overview", Technical Report UW-CSE-93-05-07, University of Washington, April 1993.

[Heid86] Heidelberger, P., "Statistical Analysis of Parallel Simulations", Proceedings of the 1986 Winter Simulation Conference, December 1986, pp. 290-295.

[Heid90] Heidelberger, P., Stone, H., "Parallel Trace Driven Cache Simulation by Time Partitioning", Proceedings of the 1990 Winter Simulation Conference, December 1990, pp. 734-737.

[Heud92] Heudin, J. C., Paneto, C., "RISC Architectures",, Chapman & Hall, 1992.

[Henn81] Hennessy, J. L., Jouppi, N. P., Baskett, F., Gill, J., "MIPS: A VLSI Processor Architecture", Proceedings of the CMU Conference on VLSI Systems and Computations, 1981, pp. 337-346.

[Henn90] Hennessy, J. L., Patterson, D. A., "Computer Architecture A Quantitative Approach", Morgan Kaufmann Publishers Inc., 1990.

[Henn91] Hennessy, J. L., Jouppi, N. P., "Computer Technology and Architecture: An evolving Interaction", IEEE Computer 24, 9, September 1991, pp. 18-29.

[Hill86] Hill, G., "Backplane Design for the T-Rack", ParSiFal Internal Document, Inmos Ltd., 1986.

[Hill89] Hill, M. D., Larus, J. R., "Cache Considerations for Programmers of Multiprocessors", Technical Report TR-CS-891, Computer Sciences Department, University of Wisconsin, Madison, November 1989.

[Hill85] Hillis, W. D., "The Connection Machine", MIT Press, 1985.

[Ho89] Ho, Y. C., "Dynamics of Discrete Event Systems" Proceedings of the IEEE, 77, 1, January 1989, pp. 3-6.

[Hoar72] Hoare, C.A.R., "Towards a theory of parallel programming", in C.A.R. Hoare and R.H. Perrot, (Editors), "Operating Systems Techniques", Academic Press, 1982.

[Hoar74] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept", Communications of the ACM, 17, 10, October 1974, pp. 549-557.

[Hoar78] Hoare, C.A.R., "Communicating Sequential Processes", Communications of the ACM, 21, 8, August 1978, pp. 666-677.

[Hoar85] Hoare, C.A.R., "Communicating Sequential Processes", Prentice Hall International, 1985.

[Hock88] Hockney, R.W., Jesshope, C.R., "Parallel Computers 2", Adam Hilger, 1988.

[Holl93] Hollingsworth, J., Miller, P. B., "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems", report accessible by anonymous ftp in grilled.cs.wisc.edu: technical_papers/w3search.ps.Z.

[Holm78] Holmes, V., "Parallel Algorithms on Multiprocessor Architectures", Ph.D. Dissertation, Computer Science Department, University of Texas, Austin, 1978.

[Hord82] Hord, R. M., "The ILLIAC IV, The First Microcomputer", Computer Science Press, Rockville, 1982.

[Hwan84] Hwang, K., Briggs, F.A., "Computer Architecture and Parallel Processing", McGraw Hill, 1984.

[Huan90] Huang, A., "Optical Computing", Lecture Notes, Sun Annual Lecture, Department of Computer Science, University of Manchester, June 1990.

[Huda90] Hudak, P., Wadler, P., "Report on the Programming Language Haskell", Version 1.0, Department of Computer Science, Yale University, April 1990.

[Huff54] Huffman, D. A.,"The Synthesis of Sequential Switching Circuits", Journal of the Franklin Institute, 257, 3, March 1954, pp. 161-190.

[Ibbe78] Ibbett, R. N., Capon, P. C., "The Development of the MU-5 Computer System", Communications of the ACM, 21, 1, January 1978, pp. 13-24.

[Ibbe82] Ibbett, R. N., "The Architecture of High Performance Computers", Macmillan, 1982.

[Inmo86] "IMS T800 Architecture", Technical Note 6, Inmos Limited, 1986.

[Inmo88] "Occam 2 Reference Manual", Prentice Hall International, 1988.

[Inmo88a] "Transputer Reference Manual", Prentice Hall Inc., 1988.

[Inmo91] "The T9000 Transputer Products Overview Manual", Inmos Limited, 1991.

[Inmo91a] "Occam2 Toolset User Manual", Inmos Limited, 1991.

[Inmo92] "Occam3 Reference Manual", Inmos Limited, 1992.

[Inmo93] "The T9000 Transputer Hardware Reference Manual", Inmos Limited, 1993.

[Inmo93a] "Networks Routers and Transputers", May, M. D., Thomson, P. W., Welch, P. H., (Editors), IOS Press, Inmos Limited, 1993.

[Inte86] "Intel iPSC System Overview", Order Number 310610-001, Intel Scientific Computers, 1986.

[Jeff82] Jefferson, D., Sowizral, H., "Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control", Technical Report N-1906-AF, RAND Corporation, December 1982.

[Jeff85] Jefferson, D., Sowizral, H., "Fast Concurrent Simulation Using the Time Warp Mechanism", Proceedings of the SCS Distributed Simulation Conference, SCS Simulation Series, 1985, pp. 63-69.

[Jeff85a] Jefferson, D., Sowizral, H., "Virtual Time", ACM Transactions on Programming Languages and Systems, 7, 3, July 1985, pp. 404-425.

[Jeff90] Jefferson, D., "Virtual Time II: Storage Management in Distributed Simulation", Proceedings of the 9th Annual ACM Symposium on Principle of Distributed Computing, August 1990, pp. 75-89.

[Jha93] Jha, V., Bagrodia, R. L., "Transparent Implementation of Conservative Algorithms in Parallel Simulation Languages", Proceedings of the 1993 Winter Simulation Conference, December 1993, pp. 677-686.

[Jone86] Jones, D. W., "Concurrent Simulation: An Alternative to Distributed Simulation", Proceedings of the 1986 Winter Simulation Conference, December 1986, pp. 417-423.

[Jone89] Jones, D. W., et al., "Experience with Concurrent Simulation", Proceedings of the 1989 Winter Simulation Conference, December 1989, pp. 756-764.

[Jone87] Jones, P., "An Extractor/Loader For the MU T-Rack", ParSiFal Internal Document PSF/MU/87/WP4/PJ/1, Department of Computer Science, University of Manchester, 1987.

[Jone88] Jones, P., Murta, A., "Support for Occam Channels via Dynamic Switching in Multi-Transputer Machines", Occam and the Transputer-Research and Applications, Editor Askew, C., IOS 1988, pp. 101-112.

[Jose90] Josephs, M. B., Udding, J. T., "Delay-Insensitive Circuits: An Algebraic Approach to their Design", Lecture Notes in Computer Science, 458, 1990, pp. 342-366.

[Jose91] Josephs, M. B,. Udding, J. T., "An Algebra for Delay-Insensitive Circuit", Proceedings of the Workshop on Computer-Aided Verification, Editors Kurshan, R., Clarke, E. M., AMS-ACM, 1991, pp. 147-175.

[Joyc87] Joyce, J., et al., "Monitoring Distributed Systems", ACM Transactions on Computer Systems, 5, 2, May 1987, pp. 121-150.

[KSR] Kendall Square Research Corporation, 170 Tracer Lane, Waltham, MA 02154-1379, USA.

[Kalu75] Kalupahana, D. J., "Causality: The Central Philosophy of Buddhism", University Press of Hawaii, 1975.

[Kane87] Kane, G., "MIPS R2000 Architecture", Prentice Hall International, 1987.

[Kate85] Katevenis, M. G. H., "Reduced Instruction Set Computer Architectures for VLSI", ACM Doctoral Dissertation Award 1984, The MIT Press, 1985.

[Kell74] Keller, R. M., "Towards a Theory of Universal Speed-Independent Modules", IEEE Transactions on Computers, 32, 1, June 1974, pp. 21-33.

[Kern88] Kerninghan, B. W., Ritchie, D. M., "The C Programming Language", Prentice Hall International, 1988.

[Kerr87] Kerridge, J., "Occam Programming, a Practical Approach", Blackwell Scientific, 1987.

[Know87] Knowles, A. E., "ParSiFal - A Parallel Simulation Facility based on the Transputer", Presented at the School of High Performance Architectures and Algorithms, Primorsko, Bulgaria, 1987.

[Know89] Knowles, A. E., Illiev, M. S., "Monitoring Facilities on the ParSiFal T-Rack", Proceedings of the ConPar'88, Cambridge University Press, 1988.

[Kogg81] Kogge, P. M., "The Architecture of Pipelined Computers", McGraw Hill Inc., 1981.

[Kona92] Konas, P., Yew, P. C., "Synchronous Parallel Discrete Event Simulation on Shared Memory Multiprocessors", Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS92), SCS, January 1992, pp. 12-21.

[Kraf79] Kraft, G. D., Toy, W. N., "Mini/Microcomputer Hardware Design", Prentice Hall International, 1979.

[Kreu86] Kreutzer, W., "System Simulation: Programming Styles and Languages", Addison Wesley, 1986.

[Kris85] Krishnamurthi, M., Chandrasekaran, U., Sheppard, S. V., "Two Approaches to the Implementation of a Distributed Simulation System", Proceedings of the 1985 Winter Simulation Conference, December 1985, pp. 435-443.

[Kuma90] Kumar, D., Harous, S., "An Approach Towards Distributed Simulation of Timed Petri Nets" Proceedings of the 1990 Winter Simulation Conference, December 1990, pp. 428-435.

[Kuma91] Kumar, D., Harous, S., "Deadlock detection and Recovery Based Distributed Simulation: A Performance Study" Proceedings of the 1991 Winter Simulation Conference, December 1991, pp. 608-617.

[Kung94] Kung, H. T., "Will Emerging High Speed Networks Provide a Solution to Parallel Architectures?", Lecture in the Sixth International School For Computer Science Researchers, July 1994, Lipari, Sicily.

[Lamp78] Lamport, L., "Time, Clocks and the Ordering of Events in Distributed Systems" Communications of the ACM, 21, 7, July 1978, pp. 558-565.

[Lau88] Lau, F. C. M., Shea, K. M., "Mapping a Process Network onto a Processor Network", in "Occam and the Transputer-Research and Applications", Editor Askew, C., IOS 1988, pp. 91-100.

[Lave83] Lavenberg, S., Muntz, R., "Performance Analysis of a Rollback Method for Distributed Simulation", Performance 1983, Elsevier Science Pub., North Holland, 1983, pp. 117-132.

[Lavi78] Lavington, S. H., "The Manchester Mark I and Atlas: A Historical Perspective", Communications of the ACM, 21, 1, January 1978, pp. 4-12.

[Lazo93] Lazowska, E. D. Statement at the U.S. House of Representatives Subcommittee on Science, Hearing on the High Performance Computing and High Speed Networking Act of 1993, H.R. 1757.

[Leu92] Leu, E., Schiper, A., "ParaRex: A programming Environment Integrating Execution Replay and Visualization" in "Environments and Tools for Parallel Scientific Computating", Editors Dongarra, J., Tourancheeau, B., Elsevier Science Publishers, 1992, pp. 155-170.

[Lin89] Lin, Y. B., Lazowska, E., "Exploiting Lookahead in Parallel Simulation" Technical Report 89-10-6, Department of Computer Science, University of Washington, 1989.

[Lin89a] Lin, Y. B., Baer, J. L., Lazowska, E., "Tailoring a Parallel Trace Driven Simulation Technique to Specific Multiprocessor Cache Coherence Protocols", Proceedings of the 1989 SCS Multiconference on

Distributed Simulation, SCS Simulation Series, March 1989, pp. 191-196.

[Lin90] Lin, Y. B., Lazowska, E., Baer, J. L., "Conservative Parallel Simulation for Systems with No Lookahead Prediction", Proceedings of the 1990 SCS Multiconference on Distributed Simulation, SCS Simulation Series, January 1990, pp. 144-149.

[Lin90a] Lin, Y. B., Lazowska, E., "Determining the Global Virtual Time in a Distributed Environment", Technical Report 90-01-02, Department of Computer Science, University of Washington, 1990.

[Lin90b] Lin, Y. B., Lazowska, E., "Reducing the State Saving Overhead for Time Warp Parallel Simulation", Technical Report 90-02-03, Department of Computer Science, University of Washington, 1990.

[Lin90c] Lin, Y. B., Lazowska, E., Bailey, M. L., "Comparing Synchronization Protocols for Parallel Logic Level Simulation", Proceedings of the 1990 International Conference on Parallel Processing, August 1990, pp. 223-227.

[Lin92] Lin, Y. B., "Memory Management Algorithms for Optimistic Parallel Simulation", Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS92), SCS, January 1992, pp. 43-52.

[LinK91] Lin, K. J., Lin, C. S., "Automatic Synthesis of Asynchronous Circuits", Proceedings of DAC, 1991, pp. 296-301.

[LinK92] Lin, K. J., Lin, C. S., "A Realization Algorithm of Asynchronous Circuits from STG", Proceedings of EDAC, 1992, pp.322-326.

[LinK92a] Lin, K. J., Lin, C. S., "On the Verification of State-Coding in STGs", Proceedings of ICCAD 1992, 1992, pp.118-122.

[Linc82] Lincoln, N. R., "Technology and Design Tradeoffs in the Creation of a Modern Supercomputer", IEEE Transactions on Computers, 31. 5. May 1982, pp. 363-376.

[Lind93] Lindsay, D., "The Limits of Chip Technology", Microprocessor Report, 7, 1, January 1993, pp. 21-24.

[Liu90] Liu, L. Z., Tropper, C., "Local Deadlock Detection in Distributed Simulations" Proceedings of the 1990 SCS Multiconference on Distributed Simulation, SCS Simulation Series, January 1990, pp. 64-69.

[Logi85] "Proposal to the Alvey Directorate for a Parallel Simulation Facility", Logica U.K Ltd., April 1985.

[Lomo88] Lomow, G., et al., "A Performance Study of Time Warp", Proceedings of the 1988 SCS Multiconference on Distributed Simulation, SCS Simulation Series, July 1988, pp. 50-55.

[Louc90] Loucks, W. M., Preiss, B. R., "The Role of Knowledge in Distributed Simulation", Proceedings of the 1990 SCS Multiconference on Distributed Simulation, SCS Simulation Series, January 1990, pp. 9-16.

[Luba88] Lubachevsky, B. D., "Bounded Lag Distributed Discrete Event Simulation", Proceedings of the 1988 SCS Multiconference on Distributed Simulation, SCS Simulation Series, July 1988, pp. 183-191.

[Luba89] Lubachevsky, B. D., "Scalability of the Bounded Lag Distributed Discrete Event Simulation", Proceedings of the 1989 SCS Multiconference on Distributed Simulation, SCS Simulation Series, March 1989, pp. 100-107.

[Luba89a] Lubachevsky, B. D., Shwartz, A., Weiss, A., "Rollback Sometimes Works ... If Filtered", Proceedings of the 1989 Winter Simulation Conference", December 1989, pp. 630-639.

[Luba89b] Lubachevsky, B. D., "Efficient Distributed Event Diven Simulations of Multiple Loop Networks", Communications of ACM, 32, 1, January 1989, pp. 111-123.

[Lump92] Lumpp, J. E., et al., "Specification and Identification of Events for Debugging and Performance Monitoring of Distributed Multiprocessor Systems", Proceedings of the 10th International Conference on Distributed Computing Systems, IEEE, 1992, pp. 476-483.

[Malo90] Malony, A, Nichols, K., "Standards Working Group Summary", in "Performance Instrumentation and Visualization", Editors Simmons, M., Koskela, R., Addison Wesley, 1990.

[Manj93] Manjikian, N., Loucks, W. M., "High Performance Parallel Logic Simulation on a Network of Workstations", Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS93), SCS, May 1993, pp. 76-84.

[Mari92] Marinescu, D. et al., "Models for Monitoring and Debugging Tools for Parallel and Distributed Software", Journal of Parallel and Distributed Computing, Academic Press, June 1992, pp. 171-184.

[Mart85] Martin, A. J., "Distributed Mutual Exclusion on a Ring of Processes", Science of Computer Programming, 5, 1985, pp. 265-276.

[Mart85a] Martin, A. J., "The design of a Self-timed circuit for Distributed Mutual Exclusion", Proceedings of the 1985 Chapel Hill Conference on VLSI, Computer Science Press, 1985., pp. 247-260.

[Mart85b] Martin, A. J., "A Delay-Insensitive Fair Arbiter", Technical Report 5193:TR:85, Computer Science Department, Caltech, 1985.

[Mart86] Martin, A. J., "Compiling Communicating Processes into Delay-Insensitive VLSI Circuits" Distributed Computing, 1, 4, April 1986, pp. 226-234.

[Mart89] Martin, A. J., "Formal Program Transformations for VLSI Circuit Synthesis", UT Year of Programming Institute on Formal Developments of Programs and Proofs, Editor Dijkstra, E.W., Addison Wesley, 1989.

[Mart89a] Martin, A. J., et al., "Design of an Asynchronous Microprocessor", Advanced Research in VLSI 1989, Proceedings of the Decennial Caltech Conference on VLSI, 1989, pp. 351-373.

[Mart89b] Martin, A. J., "The first asynchronous Microprocessor" Technical report CS-TR-89-06, Computer Science Department, Caltech, 1989.

[Mart90] Martin, A. J., "Synthesis of Asynchronous VLSI Circuits", Formal Methods for VLSI Design, Editor Staunstrup, J., North Holland, 1990.

[Matt88] Mattern, F., "Virtual Time and Global States of Distributed Systems", Proceedings of the International Workshop in Parallel and Distributed Algorithms, Gers, France, October 1988, pp. 215-226.

[May94] May, D., Invited Lecture, World Transputer Congress 1994, Como, Italy, September 1994.

[McLa92] McLaren, R., "Instrumentation and Performance Monitoring of Distributed Systems", Proceedings of the 5th Distributed Memory Computing Conference, IEEE, 1992, pp. 1180-1186.

[Mead80] Mead, C. A., Conway, L. A., "Introduction to VLSI Systems", Addison Wesley, 1980.

[Meng89] Meng, T. H. Y., Brodersen, R. W., Messerschmidtt, D. G., "Automatic Synthesis of Asynchronous Circuits from High-Level Specifications", IEEE Transactions on CAD, 8, 11, November 1989, pp. 1185-1205.

[Merr90] Merrifield, B. C., Richardson, S. B., Roberts, J. B. G., "Quantitative Studies of Discrete Event Simulation Modelling of Road Traffic", Proceedings of the 1990 SCS Multiconference on Distributed Simulation, SCS Simulation Series, January 1990, pp. 188-193.

[Mill65] Miller, R. E., "Sequential Circuits", Chapter 10, In "Switching Theory", Wiley, New York, 1965.

[Misr86] Misra, J., "Distributed Discrete-Event Simulation", ACM Computing Surveys, 18, 1, March 1986, pp. 39-65.

[Mitr84] Mitra, D., Mitrani, I., "Analysis and Optimum Performance of Two Message Passing Parallel Processors Synchronized by Rollback", Performance 1984, Elsevier Science Pub., North Holland, 1984, pp. 35-50.

[Mitc90] Mitchell, D. A. P., et al., "Inside the Transputer", Blackwell Scientific Publications, 1990.

[Miur84] Miura, K., Uchida K., "FACOM Vector Processor VP-100/VP-200", High Speed Computation, ed J. S. Kowalik , NATO ASI Series, F7, Springer Verlag, 1984.

[Mizu95] Mizuno, M., Raynal, M., Zhou, J. Z., "Sequential Consistency in Distributed Systems: Theory and Implementation", Research Report 2437, INRIA, France, May 1995.

[Mohr90] Mohr, B., "Performance Evaluation of Parallel Programs in Parallel and Distributed Systems, Proceedings of the ConPar'90 - VAPP IV, Lecture Notes in Computer Science, 475, 1990, pp. 176-187.

[Mold93] Moldovan, D. I., "Parallel Processing, From Applications to Systems", Morgan Kaufmann Publishers Inc., 1993.

[Moln83] Molnar, C. E., Fang ,T-P., "Synthesis of Reliable Speed-Independent Circuit Modules: I. General Method for Specification of Module-Environment Interaction and Derivation of a Circuit Realization", Technical Report 297, Computer Systems Laboratory, Institute for Biomedical Computing, Washington University, St. Louis, 1983.

[Mors90] Morse, K., "Parallel Distributed Simulation in Mosim", in Proceedings of the 1990 International Conference on Parallel Processing, Volume 3, August 1990, pp. 210-217.

[Muft90] Muftah, H. T., Sturgeon, R., P.,, "Distributed Discrete Event Simulation for Communication Networks", IEEE Journal on Selected Areas in Communications, 8, 9, December 1990, pp. 1723-1734.

[Mull56] Muller, D. E., Bartky, W. S., "A Theory of Asynchronous Circuits", Digital Computer laboratory 75, University of Illinois, November 1956.

[Mull57] Muller, D. E., Bartky, W. S., "A Theory of Asynchronous Circuits", Digital Computer laboratory 78, University of Illinois, March 1957.

[Murt87] Murta, A., "Tools for the Automated Configuration of a Transputer Network", MSc Thesis, University of Manchester, October 1987.

[Murt91] Murta, A. D., "Support for Transputer Based Program Development via Run Time Link Reconfiguration", Ph.D Thesis, Department of Computer Science, University of Manchester, 1991.

[Nage73] Nagel, L.W., Pederson, D. O., "SPICE (Simulation Program with Integrated Circuit Emphasis)", University of California, Berkeley, Electronics Research Laboratory, Memorandum ERL-M 382, April 1973.

[Nand92] Nandy, B., Loucks, W. M., "An Algorithm for Partitioning and Mapping Conservative Parallel Simulation onto Multicomputers", Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS92), SCS, January 1992, pp. 139-146.

[Nany94] Nanya, T., et al., "TITAC: Design of a Quasi-delay-Insensitive Microprocessor", IEEE Design and Test of Computers, 11, 2, February 1994, pp. 50-63.

[Neel87] Neelamkavil, F., "Computer Simulation and Modelling", John Wiley & Sons, 1987.

[Neto91] Neto, G. A., "Distributed Simulation Using Relaxed Timing", Technical Report UMCS-91-2-1, Department of Computer Science, University of Manchester, 1991.

[Nevi89] Nevison, C., "Discrete Event Simulation Using Occam", Proceedings of the 10th Occam User Group Technical Meeting, Enschede, April 1989, pp. 222-230.

[Nevi90] Nevison, C., "Parallel Simulation of Manufacturing Systems: Structural Factors", Proceedings of the 1990 SCS Multiconference on Distributed Simulation, SCS Simulation Series, January 1990, pp. 17-19.

[Nick95] Nicklin, S., Ph.D. Thesis, to be submitted, Department of Computer Science, University of Manchester.

[Nico84] Nicol, D. M., Reynolds, P. F., "Problem Oriented Protocol Design", Proceedings of the 1984 Winter Simulation Conference, December 1984, pp. 471-474.

[Nico88] Nicol, D. M., "Parallel Discrete Event Simulation of FCFS Stochastic Queueing Networks", SIGPLAN Notes, 23, 9, September 1988, pp. 124-137.

[Nico90] Nicol, D. M., Riffe, S. E., "A Conservative Approach to Parallelizing the Sharks World Problem", Proceedings of the 1990 Winter Simulation Conference, December 1990, pp. 186-190.

[Nico91] Nicol, D. M., Roy, S., "Parallel Simulation of Timed Petri Nets", Proceedings of the 1991 Winter Simulation Conference, December 1991, pp. 574-583.

[Nico92] Nicol, D. M., et al., "Massively Parallel Algorithms for Trace Trace Driven Simulation", Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS92), SCS, January 1992, pp. 3-11.

[Nico93] Nicol, D. M, Heidelberger P., "Parallel Algorithms for Simulating Continuous Time Markov Chains", Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS93), SCS, January 1993, pp. 3-11.

[Nico94] Nicol, D. M., Fujimoto, R., "Parallel Simulation Today", Annals of Operations Research, 53, December 1994, pp. 249-286.

[Nies88] Niessen, C., Van Berkel, C. H., Rem, M., Saeijs, R. W. J. J., "VLSI Programming and Silicon Compilation: A Novel Approach from Philips Research", Proceedings of ICCD, 1988, pp. 150-151.

[Norm93] Norman, M. G., Thanisch, P., "Models of Machines and Computation for Mapping in Multicomputers", ACM Computing Surveys, 25, 3, September 1993, pp. 263-302.

[Norr87] Norrie, C., "Supercomputers for Superproblems: An Architectural Introduction", IEEE Computer, 17, 3, March 1984.

[Nowi91] Nowick, S. M., Dill, D. L., "Synthesis of Asynchronous State Machines Using a Local Clock", Proceedings of 1991 ICCD, 1991, pp. 192-197.

[Nowi91a] Nowick, S. M., Dill, D. L., "Automatic Synthesis of Locally-Clocked Asynchronous State Machines", Proceedings of ICCAD 1991, 1991, pp. 318-321.

[Ofal] "Occam For All, A Case for Support", available at URL: http://www.hensa.ac.uk/parallel.

[Page91] Page, I., Luke, W., "Compiling Occam into FPGAs", in FPGAs, Editors Moore, W., Luk, W., Abingdon EE&CS Books, 1991, pp.271-283.

[Pars] Parsytec Computer GmbH, Juelicher Strasse 338, 52070 Aachen, Germany.

[Pars95] "Small Systems", Parsys Bulletin No. 1, Parsys Ltd, March 1995.

[Pate79] Patel, J. H., "Processor-memory Interconnection for Multiprocessors", Proceedings of the 6th International Symposium on Computer Architecture, 1979, pp. 168-177.

[Pave91] Paver, N. C., "Condition Detection in Asynchronous Pipelines", UK Patent no 9114513, October 1991.

[Pave92] Paver, N. C., et al., "Register Locking in an Asynchronous Microprocessor", Proceedings of ICCD 1992, October 1992, pp. 351-355.

[Pave92a] Paver, N. C., "Micropipelines - Implementation", Proceedings of the ACiD-WG/EXACT Workshop on Asynchronous Data Processing, Veldhoven, The Netherlands, December 1992.

[Pave94] Paver, N. C., "The Design and Implementation of an Asynchronous Microprocessor", Ph.D Thesis, Department of Computer Science, University of Manchester, 1994.

[Patt80] Patterson, D. A., Ditzel, D. R., "The Case for the Reduced Instruction Set Computer", ACM Computer Architecture News, 8, 6, October 1980, pp. 25-32.

[Peac79] Peacock, J. K., Wong, J. W., Manning, E. G., "Distributed Simulation Using a Network of Processors", Computer Networks, 3, 1, January, 1979, pp. 44-56.

[Pete91] Peterson, J. L., "Petri Net Theory and the Modelling of Systems", Prentice Hall, 1991.

[Pete93] Peterson, G. D., Chamberlain, R. D., "Exploiting Lookahead in Synchronous Parallel Simulation", Proceedings of the 1993 Winter Simulation Conference, December 1993, pp. 706-712.

[Plat] Plato, "Laws" English Translation: Loeb Classical Library, Harvard University Press.

[Poun87] Pountain, R., May, D., "A Tutorial Introduction to Occam Programming", BSP Professional Books, 1987.

[Prei89] Preiss, B. R., "The Yaddes Distributed Discrete Event Simulation Specification Language and Execution Environment", Proceedings of the 1989 SCS Multiconference on Distributed Simulation, SCS Simulation Series, March 1989, pp. 139-144.

[Prei92] Preiss, B. R., "On the Trade off between Time and Space in Optimistic Parallel Discrete Event Simulation", Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS92), SCS, January 1992, pp. 33-42.

[Prep94] Preparata, F. P., "Parallel Computation Models and Basic Techniques", Lecture in the Sixth International School For Computer Science Researchers, July 1994, Lipari, Sicily.

[Pres90] Presley, M. T., Reiher, P. L., Bellenot, S., "A Time Warp Implementation of Sharks World", Proceedings of the 1990 Winter Simulation Conference, December 1990, pp. 199-203.

[Prit91] "Principles of Modelling", Panel Session, Chair Pritsker, A. A. B., Proceedings of the 1991 Winter Simulation Conference, December 1991, pp. 1199-1208..

[Radi83] Radin, G., "The 801 Minicomputer", IBM Journal of Research and Development, 27, 3, 1983, pp. 237-246.

[Rabi94] Rabin, M., "Compiling Programs for Practical Parallel Computers", Lecture in the Sixth International School For Computer Science Researchers, July 1994, Lipari, Sicily.

[Rabi94a] Rabin, M., "Clock Construction in Fully Asynchronous Parallel Systems and PRAM Simulation", Lecture in the Sixth International School For Computer Science Researchers, July 1994, Lipari, Sicily.

[Radi92] "Discrete Event Simulation Modelling: Directions for the 90s", Panel Session, Chair Radiya, A., Proceedings of the 1992 Winter Simulation Conference, December 1992, pp. 773-782.

[Rayn95] Raynal, M., Singhal, M., "Logical Time: A Way to Capture Causality in Distributed Systems", Research Report 2472, INRIA, France, May 1995.

[Read89] Reade, C., "Elements of Functional Programming", Addison Wesley, 1989.

[Redd79] Reddaway, S.F., "The DAP approach", Infotech State of the Art Report: Supercomputers, Vol. 2, 1979, pp. 311-329.

[Reed83] Reed, D. A., "Implementing Atomic Actions on Decentralize Data", ACM Transactions on Computer Systems, 1, 1, February 1983, pp. 3-23.

[Reed87] Reed, D. A., Grunwald, D. C., "The Performance of Multicomputer Interconnection Networks", IEEE Computer, June 1987, pp. 63-73.

[Reed88] Reed, D. A., Fujimoto, R., "Multicomputer Networks: Message-Based Parallel Processing", MIT Press, 1988.

[Reed88a] Reed, D. P., Malony, A. D., McCredie, B. D., "Parallel Discrete Event Simulation Using Shared Memory", IEEE Transactions on Software Engineering, 14, 4, April 1988, pp. 541-553.

[Reed88b] Reed, D. P., Malony, A. D., "Parallel Discrete Event Simulation: The Chandy-Misra Approach", Simulation Algorithm", Proceedings of the 1988 SCS Multiconference on Distributed Simulation, SCS Simulation Series, July 1988, pp. 8-13.

[Rees85] Reese, R. M., "A Software Development Environment for Distributed Simulation", Proceedings of the SCS Distributed Simulation Conference, 1985, pp. 37-40.

[Reih90] Reiher, P.L., et al., "Cancelation Strategies in Optimistic Execution Systems", Simulation Algorithm", Proceedings of the 1990 SCS Multiconference on Distributed Simulation, SCS Simulation Series, January 1990, pp. 112-121.

[Reih90a] Reiher, P.L., Jefferson, D., "Dynamic Load Management in the Time Warp Operating System", Transactions of the Society for Computer Simulation, 7, 2, June 1990, pp. 91-120.

[Rein93] Reinhardt, S. K., et al., "The Winscosin Wind Tunnel: Virtual Prototyping of Parallel Computers", in ACM SIGMETRICS Conference on Measurement and Modelling of Computer System, 21, June 1993, pp. 48-60.

[Rem83] Rem, M, Snepscheut, J. L. A., Udding, J. T., "Trace Theory and the Definition of Hierarchical Modules", Proceedings of the 3rd Caltech Conference on VLSI, 1983, pp. 225-239.

[Reyn88] Reynolds, P. F., "A Spectrum of Options for Parallel Simulation", Technical Report IPC-TR-88-007, University of Virginia, September 1988. Also in Proceedings of the 1988 Winter Simulation Conference, December 1988, pp. 325-332.

[Reyn89] Reynolds, P. F., Weight, C. F., Filder, J. R., "Comparative Analysis of Parallel Simulation Protocols", Technical Report IPC-TR-89-011, University of Virginia, December 1989.

[Riek94] RieK, M., Tourancheau, B., Vigouroux, X. F., "Monitoring of Distributed Memory Multicomputer Programs", Technical Report UT-CS-93-204, University of Tenessee, October 1993.

[Righ89] Righter, R., Warland, J. C., "Distributed Simulation of Discrete Event Systems", Proceedings of the IEEE, 77, 1, January 1989, pp. 99-113.

[Ripl87] Ripley, B. D., "Stochastic Simulation", Wiley, 1987.

[Robe92] Roberts, J. W., editor, "Performance Evaluation and Design of Multiservice Networks", Publication of the Commission of the European Communities, Luxembourg, 1992.

[Rosc86] Roscoe, A. W., Dathi, N., "The Pursuit of Deadlock Freedom", Technical Monograph PRG-57, Programming Research Group, Computing Laboratory, Oxford University, November 1986.

[Rose94] Rosenberg, A. L., "An Overview of Mapping Problems", Lecture in Sixth International School For Computer Science Researchers, July 1994, Lipari, Sicily.

[Roth89] Rothenberg, J., "Tutorial: Artificial Intelligence and Simulation", Proceedings of the 1989 Winter Simulation Conference", December 1989, pp. 33-34.

[Rudo89] Rudolf, D. C., Reed, D. A., "Crystal: Intel iPSC/2 Operating System Instrumentation", Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications, 1989, pp. 249-252.

[Rumb91] Rumbaugh, J., et al., "Object Oriented Modelling and Design", Prentice Hall International, 1991.

[Russ78] Russell, R. M., "The CRAY-1 Computer System", Communications of the ACM, 21, 1, January 1978, pp. 63-72.

[Ruta91] Rutan, A. H., "Advances in Computer Simulation", Proceedings of the 24th Annual Simulation Symposium, IEEE 1991, pp. 2-6.

[SCS85] "Catalogue of Simulation Software", SCS Simulation, 45, 4, April 1985, pp. 196-209.

[Sado89] Sadowski, R. P., "The Simulation Process: Avoiding the Problems and Pitfalls", Proceedings of the 1989 Winter Simulation Conference", December 1989, pp. 72-79.

[Sama85] Samadi, B., "Distributed Simulation, Algorithms and Performance Analysis" Ph.D. Thesis, Department of Computer Science, University of California, Los Angeles, 1985.

[Sarg92] Sargent R. G., "Validation and Verification of Simulation Models", Proceedings of the 1992 Winter Simulation Conference", December 1992, pp. 104-114.

[Sarg94] Sargent R. G., "Validation and Verification of Simulation Models", Proceedings of the 1994 Winter Simulation Conference", December 1994, pp. 77-87.

[Sari87] Sarin, S. K., Lynch, L., "Discarding Obsolete Information in a Replicated Data Base System", IEEE Transactions on Software Engineering, 13, 1 January 1987, pp. 39-46.

[Schm88] Schmuck, F., "The Use of Efficient Broadcast in Asynchronous Distributed Systems", Technical Report TR88-927, Cornell University, 1988.

[Schn82] Schneider, F. B., "Synchronization in Distributed Programs", ACM Transactions in Programming Languages and Systems, 4, 2, April 1982, pp. 51-64.

[Schr92] Schruben, L. W., "Graphical Model Structures for Discrete Event Simulation", Proceedings of the 1992 Winter Simulation Conference, December 1992, pp. 241-245.

[Shan92] Shannon, R. E., "Introduction to Simulation", Proceedings of the 1992 Winter Simulation Conference, December 1992, pp. 65-73.

[Sega86] Segall, Z., Rudolph, L., "PIE: A Programming and Instrumentation Environment For Parallel Processing", IEEE Software, November 1985, pp. 22-37.

[Seit70] Seitz, C. L., "Asynchronous Machines Exhibiting Concurrency", Conference Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, 1970.

[Seit80] Seitz, C. L., "System Timing", Chapter 7, In [Mead80].

[Seit85] Seitz, C. L., "The Cosmic Cube", Communications of the ACM, 28, 1, January 1985, pp. 22-33.

[Shep88] Sheppard, S. V., Davis, C. K., Chandra, U., "Parallel Simulation Environments for Multiprocessor Architectures", Proceedings of the 1988 SCS Multiconference on Distributed Simulation, July 1988, pp. 109-114.

[Sieg85] Siegel, H. J., "Interconnection Networks fo Large Scale Parallel Processing: Theory and Practice", Lexington Books, 1985.

[Sifa80] Sifakis, J., "Deadlocks and Livelocks in Transition Systems", Lecture Notes in Computer Science, 88, 1980, pp. 587-599.

[Snyd82] Snyder, L., "Introduction to the Highly Parallel Computer", IEEE Computer, January 1982, pp. 47-55.

[Soko88] Sokol, L. M., Briscoe, D. P., Wieland, A. P., "MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Simulation", Proceedings of the SCS Multiconference on Distributed Simulation, SCS Simulation Series, July 1988, pp. 34-42.

[Soko89] Sokol, L. M., Stucky, B. K., Hwang, V. S., "MTW: A Control Mechenism for Parallel Discrete Simulation" Proceedings of the 1989 International Conference on Parallel Processing, August 1989, Pennsylvania, pp. 250-254.

[Soko91] Sokol, L. M., Weissman, J. B., Mutchler, P. A., "MTW: An Empirical Performance Study", Proceedings of the 1991 Winter Simulation Conference", December 1991, pp. 557-563.

[Som89] Som, T. K., Cota, B. A., Sargent R. G., "On Analysing Events to Estimate the Possible Speedup of Parallel Discrete Event Simulation", Proceedings of the 1989 Winter Simulation Conference", December 1989, pp. 729-737.

[Soul91] Soule, L., Gupta, A., "An Evaluation of the Chandy-Misra-Bryant Algorithm for Digital Logic Simulation", ACM Transactions on Modelling and Computer Simulation, 1, 4, October 1991, pp. 308-347.

[Spor93] Sporrer, C., Bauer, H., "Corolla Partitioning for Distributed Logic Simulation of VLSI Circuits", Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS93), SCS, May 1993, pp. 85-92.

[Spro86] Sproull, R. F., Sutherland, I. E., "Asynchronous Systems Volume I: Introduction", SSA 4706, Sutherland, Sproull and Associates, Inc., 1986.

[Spro94] Sproull, R. F., Sutherland, I. E., Molnar C. E., "The Counterflow Pipeline Processor Architecture", IEEE design and Test of Computers, 11, 3, March 1994, pp. 48-59.

[Stal88] Stallings, W., "Reduced Instruction Set Architecture", Proceedings of the IEEE, 76, 1, January 1988, pp. 38-55.

[Stei91] Steinman, J., "SPEEDES: Synchronous Parallel environment for Emulation and Discrete Event Simulation", Proceedings of the 5th Workshop on Parallel and Distributed Simulation (PADS91), SCS, January 1991, pp. 95-103.

[Sten90] Stenström, P., "A Survey of Cache Coherence Schemes for Multiprocessors", Computer, 23, 6, June 1990, pp. 12-24.

[Step88] Stephenson, M., Boudillet, O., "A Graphical tool for the Modeling and Manipulation of Occam Software and Transputer Hardware Topologies", in "Occam and the Transputer-Research and Applications", Editor Askew, C., IOS 1988, pp. 139-144.

[Step86] Stepney, S., "Prototype of GRAIL-Occam Screen Display", ParSiFal Internal Document PSF/GEC/WP3/86/10, 1986.

[Su89] Su, W. K., Seitz, C. L., "Variants of the Chandy-Misra Distributed Discrete-Event Simulation Algorithm", Proceedings of the 1989 SCS Multiconference on Distributed Simulation, SCS Simulation Series, March 1989, pp. 38-43.

[Sun86] Sun Microsystems Inc., "Sun 3 Architecture: A Sun Technical Report", Sun Microsystems Europe Inc., Ascot, Berkshire, 1986.

[Sun87] "The SPARC Architecture Manual", Sun Microsystems Inc., 1987.

[Suth89] Sutherland I. E., "Micropipelines", Communications of the ACM, 32, 1, January 1989, pp. 720-738.

[Suth93] Sutherland I. E., "Flashback Simulation", Research Report SunLab 93:0285, Sun Microsystems Laboratories, Inc., August 1993.

[Tadp87] Tadpole Technology PLC., "Transputer System Card TP-TSC", Preliminary Product Description, Tadpole Technology PLC, 1987.

[Theo91] Theodoropoulos, G, "Specification of Replicated Structures in a Graphical Environment for a Transputer based Machine", MSc Thesis, University of Manchester, October 1991.

[Theo94] Theodoropoulos, G., "An occam model of the AMULET1", In Proceedings of the AMULET Modelling Workshop, Windermere, Cumbria, England, July 1994.

[Theo94a] Theodoropoulos, G., Woods J.V., "Building Parallel Distributed Models for Asynchronous Computer Architectures", Proceedings of the World Transputer Congress 1994, Como, September 1994, pp. 285-301.

[Theo94b] Theodoropoulos, G., Woods J.V., "Distributed Simulation of Asynchronous Computer Architectures: The Program Driven Conservative Approach", Proceedings of the European Simulation Symposium 1994, Volume 2, Istanbul, Turkey, October 1994, pp. 230-234.

[Theo94c] Theodoropoulos, G., West, A., "Graphical Configuration of Transputer Systems: The Graphical Configuration Assistant", Proceedings of the 1994 Transputer Research and Applications Conference (NATUG-7), Athens, Georgia, October 1994.

[Theo95] Theodoropoulos, G., Woods J.V., "Analysing the Timing Error in Distributed Simulations of Asynchronous Computer Architectures", Eurosim Congress '95, Vienna, Austria, September 1995, to appear.

[Theo95a] Theodoropoulos, G., Woods J.V., "Dealing with Time Modelling Problems in Parallel Models of Asynchronous Computer Architectures", World Transputer Congress 1995, Harrogate, England, September 1995, to appear.

[Theo95b] Theodoropoulos, G., Woods J.V., "Simulating Asynchronous Architectures on Transputer Networks", 4th IEEE Euromicro Workshop On Parallel And Distributed Processing, Braga, Portugal, January 1996, to appear.

[Thes90] Thesen, A., Travis, L. E., "Introduction to Simulation", Proceedings of the 1990 Winter Simulation Conference, December 1990, pp. 14-21.

[Thom91] Thomas, G. S., Zahorjan, J., "Parallel Simulation of Performance Petri Nets: Extending the Domain of Parallel Simulation", Proceedings of the 1991 Winter Simulation Conference, December 1991, pp. 564-573.

[Tink89] Tinker, P. A., Agre, J. R., "Object Creation, Messaging, and State Manipulation in an Object Oriented Time Warp System", Proceedings of the 1989 SCS Multiconference on Distributed Simulation, SCS Simulation Series, March 1989, pp. 79-84.

[Trel82] Treleaven, P.C., Brownbridge, D.R., Hopkins, R.P., "Data-Driven and Demand-Driven Computer Architecture", ACM Computing Surveys, 14, 1, January 1982, pp. 93-143.

[Turn92] Turner, S, Xu, M., "Performance Evaluation of the bounded Time Warp Algorithm", Proceedings of the 6th Workshop on Parallel and Distributed Simulation, SCS Simulation Series, January 1992, pp. 117-128.

[Unge86] Unger, B., et al., "A Distributed Software Prototyping and Simulation Environment: Jade", Proceedings of SCS Multiconference on Intelligent Simulation Environments, SCS Simulation Series, 1986, pp. 63-71.

[Unge89] Unger, B., "The Impact of Parallel Processing", in "How Technology Limits Simulation Methodology", Panel Session, Chair Pegden, C. D., Proceedings of the 1989 Winter Simulation Conference, December 1989, pp. 686-691.

[Unge69] Unger, S. H., "Asynchronous Sequential Switching Circuits", Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[VLSI90] "Acorn Risc Machine (ARM) Family Data Manual", VLSI Technology Inc., Prentice Hall International, 1990.

[VaBe88] Van Berkel, C. H., Rem, M., Saeijs R. W. J. J., "VLSI Programming", Proceedings of ICCD, 1988, pp. 152-156.

[VaBe88a] Van Berkel, C. H., Rem, M., Saeijs, R. W. J. J., "Compilation of Communicating Processes into Delay-Insensitive Circuits", Proceedings of ICCD, 1988, pp. 157-162.

[VaBe91] Van Berkel, C. H., Kessels, J., Roncken, M., Saeijs, R. W. J. J, Schalij, F., "The VLSI-Programming Langauge Tangram and its Translation into Handshake Circuits", Proceedings of EDAC, 1991, pp. 384-389.

[VaBe92] Van Berkel, C. H., "Handshake circuits: An Intermediary Between Communicating Processes and VLSI", Ph.D thesis, Eindhoven University of Technology, 1992.

[Vanb90] Vanbekbergen, P., et al., "Optimized Synthesis of Asynchronous Control Circuits from Graph-Theoretic Specifications" Proceedings of ICCAD 1990, 1990, pp. 184-187.

[Venk86] Venkatesh, K., Radhakrishnan, T., Li, H. F., "Discrete Event Simulation in a Distributed System", IEEE COMPSAC 1986, IEEE Computer Society Press, pp. 123-129.

[Wagn89] Wagner, D. B., Lazowska, E. D., "Parallel Simulation of Queueing Networks: Limitations and Potentials", Proceedings of the International Conference on Measurement and Modeling of Computer Systems, Berkeley, USA, May 1989, pp. 146-155.

[Warr88] Warren, H. D., Haridi, S., "The Data Diffusion Machine - A scalable Shared Virtual Memory Architecture for Parallel Execution of Logical Programs", Proceedings of the 1988 International Conference on Fifth Generation Computer Systems, 1988, pp. 943-952.

[Waym89] Wayman, R., "Transputer Development Systems", in Transputer Applications, Harp, G. (Editor), Pitman, 1989, pp. 31-83.

[Weic84] Weicker, R. P., "Dhrystone, A Synthetic Systems Programming Benchmark", Communications of the ACM, 27, 10, October 1984, pp. 1013-1030.

[Welc87] Welch, P. H., "Emulating Digital Logic Using Transputer Networks (Very High Parallelism = Simplicity = Performance)", Lecture Notes in Computer Science, 258, (PARLE'87), 1987 pp. 357-373.

[Welc93] Welch, P. H., Justo, G., Willock, C., "High-Level Paradigms for Deadlock-Free High-Performance Systems", Proceedings of the World Transputer Congress 1993, September 1993, pp. 981-1004.

[Wern84] Werner, J., Beresford, R., "A System Engineer's Guide to Simulators", VLSI Design, February 1984, pp. 27-31.

[Whar92] Wharton, J., "Why RISC is Doomed", Microprocessor Report, 6, 11, August 1992, , pp. 14-17.

[Whit89] Whitner, R., and Balci, O., "Guidelines for Selecting and Using Simulation Model Verification Techniques", Proceedings of the 1989 Winter Simulation Conference, December 1989, pp. 559-568.

[Wiel89] Wieland, F., et al., "Distributed Compat Simulation in Time Warp: The Model and its Performance", Proceedings of the 1989 SCS Multiconference on Distributed Simulation, SCS Simulation Series, March 1989, pp. 14-20.

[Wile87] Wiley, P., "A Parallel Architecture Comes of Age at Last", IEEE Spectrum, June 1987, pp. 46-50.

[Wirt77] Wirth, N., "Modula: A Language for Modular Multiprogramming", Software Practice and Experience, 7, 1, January 1977, pp. 3-35.

[Wood94] Wood, K. R., Turner, S. J., "A Generalized Carrier-Null Method for Conservative Parallel Simulation", Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS94), SCS, July 1994, pp. 50-57.

[WooJ94] Woods, J. V., Private Communication, 1994.

[Wulf72] Wulf, W.A., Bell, C.G., "C.mmp - A multi-mini-processor", Proceedings of the AFIPS Fall Joint Computing Conference, 41, part 2, 1972, pp. 765-777.

[Wyat83] Wyatt, D. L., Sheppard, S., Young, R. E., "An Experiment in Microprocessor-Based Distributed Digital Simulation", Proceedings of the 1983 Winter Simulation Conference, December 1983, pp. 271-277.

[Wyat84] Wyatt, D. L., Sheppard, S., "A Language Directed Distributed Discrete Simulation System", Proceedings of the 1984 Winter Simulation Conference, December 1984, pp. 463-464.

[Wyat85] Wyatt, D. L., “Simulation Programming on a Distributed System: A Preprocessor Approach”, Proceedings of the SCS Distributed Simulation Conference, SCS Simulation Series, 1985, pp. 32-36.

[Xu89] Xu, M., Pin, N., “An Irregular Distributed Simulation Problem with a Dynamic Logical Process Structure”, Proceedings of the 11th Occam User Group Technical Meeting, Edinburgh, September 1989, pp. 213-221.

[Yako92] Yakovlev, A. V., “On Limitations and Extensions of STG Model for Designing Asynchronous Control Circuits”, Proceedings of ICCD 1992, pp. 396-400.

[Yovi92] Yovits, M. C., editor, “Advances in Computers”, Vol. 35, Academic Press, 1992.

[Yu89] Yu, Q., Towsley, D., Heidelberger, P., “Time Drivel Parallel Simulation of Multistage Interconnection Networks”, Proceedings of the 1989 SCS Multiconference on Distributed Simulation, SCS Simulation Series, March 1989, pp. 191-196

[Yuce92] Yucesan, E., Jacobson, S., “Building Correct Simulation Models is Difficult”, Proceedings of the 1992 Winter Simulation Conference, December 1992, pp. 783-789.

[Yun92] Yun, K. Y., Dill, D. L., Nowick, S. M., “Synthesis of 3D Asynchronous State Machines”, Proceedings of ICCD 1992, 1992, pp. 346-350.

[Yun92a] Yun, K. Y., Dill, D. L., Nowick, S. M., “Practical Generalizations of Asynchronous State Machines”, Technical Report CSL-TR-92-544, Computer Systems Laboratory, Stanford University, July 1992.

[Zaba92] Zabala, E., Taylor, R., “Process and Processor Interaction”, in “Environments and Tools for Parallel Scientific Computing”, Editors Dongarra, J., Tourancheeau, B., Elsevier Science Publishers, 1992, pp. 55-72.

[Zeig76] Zeigler, B. P., “Theory of Modelling and Simulation”, John Wiley and Sons, 1976.

[Zeig90] Zeigler, B. P., “Object Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems”, Academic Press, 1990.

[Zhan89] Zhang, G., Zeigler, B. P., “DEVS-Scheme Supported Mapping of Hierarchical Models onto Multiple Processor Systems”, Proceedings of the 1989 SCS Multiconference on Distributed Simulation, SCS Simulation Series, March 1989, pp. 57-60.