On the Distribution of Control in Asynchronous Processor Architectures

Vinod Eugene Francis Rebello

Doctor of Philosophy

The University of Edinburgh

1996

To my parents

Peter and Gemma

Abstract

The effective performance of computer systems is to a large measure determined by the synergy between the processor architecture, the instruction set and the compiler. In the past, the sequencing of information within processor architectures has normally been synchronous: controlled centrally by a clock. However, this global signal could possibly limit the future gains in performance that can potentially be achieved through improvements in implementation technology.

This thesis investigates the effects of relaxing this strict synchrony by distributing control within processor architectures through the use of a novel asynchronous design model known as a *micronet*. The impact of asynchronous control on the performance of a RISC-style processor is explored at different levels. Firstly, improvements in the performance of individual instructions by exploiting actual run-time behaviours are demonstrated. Secondly, it is shown that micronets are able to exploit further (both spatial and temporal) instruction-level parallelism (ILP) efficiently through the distribution of control to datapath resources. Finally, exposing fine-grain concurrency within a datapath can only be of benefit to a computer system if it can easily be exploited by the compiler. Although compilers for micronet-based asynchronous processors may be considered to be more complex than their synchronous counterparts, it is shown that the variable execution time of an instruction does not adversely affect the compiler's ability to schedule code efficiently. In conclusion, the modelling of a processor's datapath as a micronet permits the exploitation of both fine-grain ILP and actual run-time delays, thus leading to the efficient utilisation of functional units and in turn resulting in an improvement in overall system performance.

Acknowledgements

I am indebted to my supervisor, D. K. Arvind, for his continuous support, encouragement and advice throughout my research.

Thanks to the MAP Group for our fruitful discussions; to the Edinburgh Parallel Computing Centre (EPCC) for access to the ME*i*KO Computing Surface and their technical support; and to the Department of Computer Science for providing all the "essentials" for this work.

Most of all, a big special thank you to my parents and all my friends who shared in my trials.

Finally, this work was funded by a research studentship from the UK Science and Engineering Research Council.

Muito obrigado para todos!

Declaration

This thesis was composed by myself and the work reported herein is my own except where indicated. Some of the material in this thesis has already been published in:

- D. K. Arvind and V. E. F. Rebello. Instruction-level parallelism in asynchronous processor architectures. In M. Moonen and F. Catthoor, editors, *The Proceedings of the 3rd International Workshop on Algorithms and Parallel VLSI Architectures*, pages 203–215, Leuven, Belgium, August 1994. Elsevier Science Publishers.

- D. K. Arvind and V. E. F. Rebello. On the performance evaluation of asynchronous processor architectures. In P. Dowd and E. Gelenbe, editors, *The Proceedings of the 3rd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'95)*, pages 100–105, Durham, NC, USA, January 1995. IEEE Computer Society Press.

- D. K. Arvind, R. D. Mullins and V. E. F. Rebello. Micronets: A model for decentralising control in asynchronous processor architectures. In M. B. Josephs, editor, *The Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, pages 190–199, London, UK, May 1995. IEEE Computer Society Press.

- D. K. Arvind and V. E. F. Rebello. Static scheduling of instructions on micronet-based asynchronous processors. In *The Proceedings of the 2nd International Symposium on Advanced Research on Asynchronous Circuits and Systems (ASYNC'96)*, pages 80–91, Aizu Wakamatsu City, Japan. March 1996. IEEE Computer Society Press.

Vinod E. F. Rebello

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

> *"In analysing the functions of the contemplated device, ... the logical control of the device, that is the proper sequencing of its operations, can be most efficiently carried out by a central organ."*
> John von Neumann, First Draft of a report on the EDVAC (1945)

It has long been realised that the implementation technology has influenced developments in processor architectures. As a case in point, the advent of VLSI technology in the early 1980s (together with mature optimising compilers) led to the reassessment of complex instruction sets, and resulted in the development of RISC architectures [71] [86]. The designers of these processors also paid close attention to the interactions between the compiler, the instruction set, and the processor architecture. Reducing the number and formats of instructions made the architecture considerably simpler compared to existing designs, with streamlined datapaths effectively shifting complexity from the hardware to the compiler.

Improvements in transistor speed have brought improvement in system performance [121], and it has been assumed that such progress would continue virtually unhindered. However, designers have now been forced to consider a domain previously taken for granted – the influence of the control paradigm on the rest of the system. From around 1945, conventional wisdom has advocated the use of a centralised clock to sequence information correctly within a processor architecture. Unfortunately, the ability to sustain this design style as systems become larger, faster and more complex, is under pressure from a number of directions, related to the global clock as well as the speed and scale of the new systems [115,140,147,150,175].

Given the developments in technology, and contradicting John von Neumann, centralised control can lead to inefficient behaviour. Events in synchronous processors are recognised at regular, pre-determined intervals. In typical designs, there are idle periods between events and the next clock tick. Of course, this wastage could be reduced by increasing the clock frequency, but the benefit of such a policy is diminished by problems of increased control complexity, clock skewing and noise. Furthermore, the clock's very presence is likely to limit future gains in performance which may potentially be achieved by improvements in VLSI technology. The maximum speed of this clock signal is a conservative estimate for reliable operations, which considers worst-case delays in the critical path. In practice, even this estimate may not be met due to variations in fabrication and environmental parameters. The propagation delays along clock distribution lines may become a significant proportion of the clock period, and mitigating their effect at higher frequencies would be at significant design costs [42]. These inefficiencies are further exacerbated by scaling of transistor sizes [115] [140] [147]. Another issue is the difficulty in separating the logical and temporal aspects of synchronous circuits. Accurate estimations of timings of synchronous processors and abstracting them from the logical design is difficult. This has been one of the limiting factors in the automatic synthesis of synchronous processors. All of these drawbacks have led to a renewed interest in an alternative control strategy which relaxes the strict synchrony imposed by the centralised clock by removing it altogether.

Asynchronous design is not new, in fact early computers did incorporate asynchronous methods which were later abandoned in favour of the easier synchronous style. Lately, a restricted form of asynchrony known as self-timing is being considered which avoids timing-related problems by enforcing a simple communication protocol [150]. This protocol acts like a local clock which synchronises components within a circuit, but neither relies on specific time intervals nor extends homogeneously to the entire circuit as a synchronous clock does. The correct operation of self-timed systems is independent of delays, enabling systems to cope with changes due to data dependencies or environmental variations. This robustness is achieved at the price of local handshaking protocols. Therefore, in order to exploit the performance benefits of asynchrony over synchronous control, the average delay of the components together with overheads of self-timed control should be less than the sum of the worst-case delay and overheads of synchronous control. However, it was not the potential performance advantage of self-timed circuits which first attracted processor designers.

Self-timed circuits offer a number of other advantages over their synchronous equivalents (as discussed in [66] [106]) and, for example, have proved attractive for low power circuit design and automated synthesis. Asynchronous microprocessor designs (which have been built) have either concentrated on their formal synthesis [37] [110] or just their feasibility [143], with limited emphasis on their performance or efficient operation. One exception is the AMULET project [56]: an asynchronous implementation of a previous synchronous design, although the emphasis has primarily been on low power consumption. The performance evaluation of asynchronous processors is still in its infancy. Only recently have designs begun to take architectural considerations into account, e.g. Counterflow [157] and Fred [142], and investigate issues such as Instruction-Level Parallelism (ILP).

Synchronous architectures exploit ILP at a considerable cost in terms of control overheads. Also, this centralised control regime forces complex designs to operate below their technological best by always assuming worst-case behaviour. The benefit, however, is that the computational model uses fixed delays thus leading to a deterministic behaviour of the architecture. This benefits the compilers in predicting the state of the machine for efficient code generation and scheduling. Therefore, forcing operations to complete within a fixed period of time simplifies the cost of sequencing operations. In contrast, under asynchronous control, operations take only as long as is necessary; even the execution times of identical instructions may vary. This, in turn, may have an adverse effect on efficient code generation and scheduling. However, note that exploiting concurrent behaviour is more efficient under distributed control, whereas synchronising operations or making them take place sequentially increases the control complexity in an asynchronous environment.

1.1 In this Thesis

The RISC approach exploited the synergy in the interactions between the three domains – the compiler, the processor architecture and the implementation technology. The work described in this thesis builds on this theme and investigates the design of effective computer systems in the light of progress in each of these domains; in particular, the efficient exploitation of ILP in fully asynchronous general-purpose processor architectures.

There has also been an important trend in identifying and exploiting concurrency in programs which are written in languages without explicit parallel constructs. The concurrency is exposed in different stages of descending levels of granularity: between basic blocks, between instructions within the same block, and even within the instructions themselves. In general, concurrency between basic blocks can be teased out by the compiler without an intimate knowledge of the underlying processor. However, for effective exploitation of concurrency at a finer detail of granularity, it is profitable to consider the interactions between the compiler and the processor, and the processor and the implementation technology, respectively. Increased performance through the exploitation of ILP is a key feature of modern synchronous RISC processor architectures. However this approach is limited not only by the available parallelism within programs, but also by the cost effectiveness of designing processors with centralised control to exploit ILP.

This thesis studies the influence of a fully asynchronous control paradigm on the design and performance of RISC-like processor architectures. The justification for doing this is the following observation. The clock period of a synchronous processor is determined *a priori* by the speed of the slowest component, and takes into account the worst-case execution and propagation times and the worst-case operating conditions. In contrast, the performance of an asynchronous processor is determined by the actual operational timing characteristics of the components (effectively average delays) plus the overheads due to self-timed control. Furthermore, a more significant and important consequence of an asynchronous control paradigm is the ability to exploit fine-grain concurrency efficiently at the instruction level.

Processors can be divided into two parts – the datapath and the control. In synchronous designs, the centralised control performs the dual functions of timekeeping and sequencing of operations within the datapath. Timekeeping is now redundant in an asynchronous processor, thereby reducing the rôle of the centralised control to just sequencing instructions. An asynchronous datapath can be modelled and implemented as a *micronet*. Defined as a network of elastic micropipelines [158], it allows for a greater degree of fine-grained concurrency to be exploited, both between and within instructions, which would otherwise be quite expensive to achieve in an equivalent synchronous design. In a traditional synchronous datapath, the centralised control forces each instruction to go through all of the stages regardless of the need to do so (in effect a single pipeline), with the time spent in each stage being determined by the clock period. In a micronet, each program instruction spends time only in the relevant stages and for just as long as is necessary. Furthermore, different program instructions may execute concurrently within the same stage. A synchronous pipelined processor for exploiting ILP has to incur additional control overheads, e.g. [40] [42] [118]. In contrast, it will be demonstrated that as a consequence of asynchronous control, implemented using a micronet, ILP can be achieved implicitly without extra costs. This is because the control is now decentralised and distributed amongst the communicating functional units which operate concurrently. Micronets are easy to implement in CMOS VLSI technology [126], and at the same time, as will be shown, they offer a good target for an optimising compiler which can exploit the available concurrency between and within instructions.

1.2 Thesis Outline

The contents of each of the remaining chapters are summarised as follows:

Chapter 2 highlights the inefficiencies in current synchronous designs and introduces a particular field of asynchronous design known as self-timed circuits as a methodology to overcome these problems. How self-timed circuits communicate while being insensitive to varying delays and the advantages of these types of circuits are also discussed. This chapter then sets out the objectives and goals of this thesis in the context of current related work and opinion, and introduces an efficient structure for distributed asynchronous control called a *micronet*.

Chapter 3 – The performance of an asynchronous system is ultimately determined by the dynamic interaction amongst components within the system. Furthermore, the temporal behaviour of current VLSI systems is being increasingly influenced by propagation delays which themselves can only really be determined after layout has taken place. Therefore, evaluating the performance of these systems via analytical methods is difficult. Estimating program performance via logic simulation is impractical due to the amount of CPU time required. However, an application such as an asynchronous processor is particularly well suited to parallel discrete event simulation (PDES) due the inherent parallelism afforded by the distribution of control.

This chapter describes PEPSÉ, a simulation platform on a network of transputers [79] for evaluating the performance of asynchronous processor architectures. The architectures can be modelled at various levels of abstraction in the programming language Occam2 [78]. Occam2 is based on the process model of computing in which a system can be described as a collection of concurrent processes which communicate with each other asynchronously through channels. The semantics of Occam2 capture the behaviour of asynchronous circuits naturally. The underlying timekeeping mechanism in PEPSÉ is based on a parallel asynchronous simulation algorithm described in [8]. The asynchronous nature of this algorithm efficiently simulates the class of architectures under investigation compared to time-driven simulations.

Chapter 4 investigates, through simulation, the improvements in instruction execution times of an asynchronously-controlled processor when compared to an equivalent synchronously-controlled one. This study only exploits the average delays of the functional units in the self-timed case to reduce the execution times of the individual instructions. Results show that shorter execution times can be achieved under micronet control. Taking datapath pipelining into account at this stage is considered inappropriate since pipelining increases both the control complexity and the instruction latency.

Chapter 5 concentrates on the use of micronets to exploit ILP, which also requires a number of control issues resulting from data and structural dependencies between instructions to be resolved efficiently. Suitable metrics are introduced for measuring this and the performance of asynchronous processors. The exploitation of ILP is analysed through a number of refinements made to the Micronet-based Asynchronous Processor (MAP) design of the previous chapter. Centralised control is progressively distributed to the functional units and the effects on the overall performance of simple test programs are recorded. Results show that a micronet-based datapath allows a greater degree of fine-grained concurrency to be exploited.

Chapter 6 discusses the influences of the asynchronous control paradigm on the compiler of a micronet-based architecture. It is important to demonstrate that the asynchronous processor is still a good target for a parallelising compiler. The back-end of a compiler has two machine-dependent tasks, namely to generate code and schedule the instructions. It will be demonstrated that the local scheduling of a basic block can be performed efficiently.

A micronet compiler is unable to predict the exact behaviour of the architecture for the execution of a given set of instructions. This is because the execution times may vary due to data dependent operations and to interactions between executing instructions competing for the same resources. However, an instruction schedule based on worst-case operational behaviour can provide an upper bound on the program's execution time. This is useful since, generally, compilation is carried out once and programs are run many times. Further performance improvement may be obtained at run-time, to exploit the actual and data dependent delays, by fine-tuning the instruction schedule dynamically.

Chapter 7 draws conclusions and includes discussions on the implications for processor design and future work. A glossary of terms appears in Appendix A.

Chapter 2

Towards an Asynchronous Control Paradigm

2.1 Introduction

This chapter focuses on a previously implicit factor in computer system design called the **control paradigm**, and examines the motivation behind investigating the use of an *asynchronous* control paradigm in RISC processor architectures. Synchronous controls have been the norm in processor designs. But lately, there has been a resurgence in the use of asynchronous design styles where instead of using a global clock to regulate operations and communicate information at fixed intervals, operations take place autonomously and communication takes place at arbitrary times whenever information transfer is necessary. Some of the motivation behind this interest has been due to the difficulties envisaged in synchronous VLSI design. This chapter outlines these concerns, the inefficiencies in synchronous control and the advantages of asynchrony. More importantly, the effect of the control paradigm on the exploitation of instruction-level parallelism in the traditional view of processor datapaths is discussed. It is believed that the asynchronous approach can provide a more efficient design style for processor architectures.

2.2 System Design

The design of a well integrated RISC microprocessor system should consider the relationships between the different aspects of the system. The RISC experience highlighted the need to consider the interactions between the implementation technology, the processor architecture (which efficiently implements a given instruction set) and the compiler. The shift from CISC to RISC architectures took advantage of maturing optimising compilers and improved VLSI technology. The implementation technology has continued to play a significant part in improving system performance of these architectures. However, current advances are adversely affecting the synchronous control paradigm's ability to exploit the potential performance gains *efficiently*. In synchronous processors, while diminishing feature sizes and increasing clock speeds bring better performance, they are achieved at a significant cost and design effort. Even the underlying efficiency of this improvement is falling due, for example, to increases in power consumption and the greater proportion of the clock period which needs to be set aside to account for the side effects of technological advances.

2.3 Implementation Technology and a Synchronous Control Paradigm

The improvements in integrated circuit technology pose new constraints on the design of synchronous processors. *Control* management is characterised by a global synchronising signal or clock to make all of the components in the design communicate correctly, i.e. the clock controls both the sequencing and the timing within circuits. Though not always appreciated, this global clock can significantly limit the performance in a large system. This is due, in part, to a number of factors. Firstly, the clock period needs to account for some underlying physical characteristics of VLSI circuits related to the cost of distributing the clock and the loading on clock buffers. Thus, part of the clock period must be set aside to allow for clock skew. Secondly, the clock speed must be a conservative worst case, not only in terms of the component's critical-path delay, but also of fabrication and environmental parameters (if the chip is to operate reliably). Finally, transistors switch virtually simultaneously, causing the power supply inductance to become a more significant limitation on switching speed.

2.3.1 Clock Skew

Some components in a synchronous design may see the global clock signal change before others because of variations in propagation delays (due to differences in track length and loading) along the clock distribution lines. This discrepancy, known as clock skew, means that the effective computation time available is less than the clock period. In order to ensure correct operation, the clock period must be increased which implies a limit on the maximum clock frequency. Reducing the clock skew requires detailed analysis of the load on the clock signal and careful design of the clock drivers, which incurs significant cost and design effort [42].

2.3.2 Other Limits on the Clock Frequency

Synchronous designs are optimised for worst-case conditions. The clock period (and hence maximum frequency) is limited by the operation that takes the longest time to complete which is determined by the slowest component, its slowest operation, its worst-case data inputs and the worst-case operating conditions (i.e. supply voltage, temperature and fabrication process). Designers try to reduce this delay by speeding up the component's logic for degenerate data input and by balancing component delays. However in synchronous designs, effort must be invested in analysing logic which might be rarely used, in order to find and speed up the critical path.

Furthermore, the slowest operation may not even be required in a particular clock period. There has been some work on varying the period of the clock dynamically depending on the operation [39]. An alternative approach is the incorporation of multiple frequency clocks into designs (generally derived from a single clock), which requires analogue circuitry i.e. phase-locked loops. Both these approaches are difficult and expensive for the high clock frequencies at which modern processors operate.

2.3.3 Power Consumption

Power consumption is increasingly becoming an important factor in processor design. In CMOS circuits, the majority of power is consumed during the switching of gates. Most of them take place at clock transitions in synchronous designs causing peaks of power consumption and leading to voltage drops due the inductance of the power supply. (Extreme variations can cause the system to malfunction.) Also, periodic high currents on a chip can cause electromigration: the force of the moving electrons hitting metal atoms causing deformations and breaks in the metal [159]. Designers resort to using decoupling capacitors, many power pins and wide power rails to reduce these effects at the expense of packaging costs (e.g. gold is now being used in some designs for bond wires, pads and power distribution rails [65] [83]). For example, the DEC 21064 Alpha chip requires 138 power and ground pins to supply its 30W power requirement and the 43A peak switching current drawn by the clock [42] [114].

Synchronous systems distribute the clock to all of the components which means that they consume power whether they are doing useful work or not. Selective disabling of the clock signal adds complexity to the clock buffers and exacerbates the clock distribution problem, especially at high clock frequencies. Power consumption can also be reduced by decreasing the power supply voltage. However, since transistor threshold voltages must scale down with supply voltage, it may become increasingly difficult to make transistors with small enough thresholds.

If the supply voltage is not reduced in proportion to the decrease in feature size, then the power consumption per unit area will increase. Together with the fact that in CMOS the power dissipated is proportional to the frequency of the clock [175], it seems likely that the upward trend in power consumption (especially of microprocessors) will continue. Eventually, one might envisage performance being limited by heat dissipation unless cost effective techniques can be found. Removing heat from chips will become increasingly difficult and therefore expensive. Solid (passive) heat sinks to cope with even moderate power levels (50W to 100W) are large and require significant air-flow. For higher ranges, more active devices become necessary, e.g. a thermosiphon [65] [83].

2.3.4 Shrinking Geometries

As the physical size of transistors and connections, known as the feature size, is scaled down, therefore allowing a larger number of more complex and faster circuits to be fabricated on a single chip, the problems associated with synchronous design (clock skew and power consumption) will become increasingly significant [115] [140].

The ability of synchronous designs to take advantage of these smaller, faster devices is being hindered by timing delays in the interconnection layers [147]. In VLSI circuits, wiring delays are approaching a significant proportion of switching delays and can no longer be ignored. Scaling exacerbates these problems: since systems contain more circuits, global signals have to travel longer distances relative to transistor sizes. This may mean proportionally reducing the clock period, which would result in inefficient operation of the system.

The Effects of Scaling

It is informative to observe how a circuit's operation is affected when its spatial dimensions are scaled down by a factor α [175]. (Assume that the circuit's operating voltage is divided by α too. This keeps both the electric fields on the chip and the power dissipation per unit area constant.)

The propagation of electrical signals through a circuit is attenuated by two delays: in the channels of transistors and in the wires. The former, often called the transit time τ, is the time taken by charge carriers to "cross" the electric field in the channel. Since this field is unaffected by scaling, the transit time is divided by α (the channel becomes shorter), resulting in faster transistors. The delay that signals encounter in wires is determined by the rate at which a voltage presented at one end of a wire equalises across the whole wire. For a wire of length l, this is proportional to $R.C.l^2$, where R and C are the resistance and capacitance of the wire per unit length, respectively. When scaled down, R is increased by a factor of α^2, C is unaffected, and l is divided by α. Consequently, the wire delay does not change under scaling. But since the transit time is shortened, the wire delay increases relative to the transit time. If the correct functioning of a circuit depends on the relation between these delays, then the shrunk version may not function correctly any longer.

Delays in short wires are much shorter than delays in transistors. For small chip areas the wire delay may, therefore, be ignored. Such an area is known as an isochronic or equipotential region [106] [150]. By dividing a circuit into sufficiently small subcircuits and realising each subcircuit in an isochronic region, only the wire delays of the connections between different subcircuits need to be taken into account.

Locality

It is clear that since gate delays decrease with scaling, whereas interconnection delays remain constant, eventually the speed at which a circuit can operate will be dominated by interconnect delays rather than device delays. However, the situation is actually somewhat worse than the above consideration implies, due to a factor known as *stuffing*. This means that the lengths of the interconnections do not scale down with the inverse of the scaling factor, as was assumed. In practice, as the complexity of the circuit increases, the distance over which interconnections must be maintained on a chip of fixed area may stay roughly constant. It has been argued from statistical considerations [89] that a good approximation to the maximum length L_{max} of interconnection required is given by

$$L_{max} = \frac{A^{1/2}}{2}$$

where A represents the area of the chip. Therefore, the average interconnection delay may actually increase. If scaling occurs and the chip size is also increased, then the interconnect problem is further exacerbated. When the delay time of the circuit depends largely on the interconnection delay (instead of the logic gate delay), minimal and local interconnections will become an essential factor for an effective realisation of the VLSI circuit [96].

2.3.5 Design Difficulties

The clock in a synchronous circuit can be a source of both transient and permanent errors [150]. Even when modules communicate correctly under ideal or typical conditions, timing problems can still arise. A change in clock speed, caused by processing or the environment, can make the system fail even if a conservative one is chosen. For example, it could exaggerate clock skew and require increased setup and hold times. For systems running at their maximum clock frequency, this means reduced reliability. Overcoming these timing problems in synchronous designs is far from trivial and is one of the causes of devices being either slow, unreliable, or not working at all.

Thus, improvements in IC technology pose new constraints on the design of synchronous processors and since the clock has to be proportionally reduced this results in an inefficient operation of the system. The use of global clock signals also affects other areas of the design process. In synchronous designs the timing of a circuit, being fundamental to its correct operation, is one of the most difficult parameters to abstract from the logical design. Designers must always be aware of the performance of the hardware implementation in order to verify its operational correctness. Also, as a consequence of the automated layout of circuits, the designer has less control over the exact placement of global signal lines. Therefore, the true performance of these designs is difficult to estimate accurately. For example, in the design of the DEC Alpha 21064, designers had to use post-layout simulations and three-dimensional representations of the results to evaluate the clock skew across the chip [42]. This violates the hierarchical approach to design by making it more difficult to abstract away from the electrical characteristics of the VLSI implementation [147].

2.4 Asynchronous Design – A Solution?

Asynchronous design attempts to solve some or all of the problems described previously. Asynchronous circuits have no global clock, and therefore are free from global synchronisation operational and design problems. Asynchronous circuits can be based on different timing models. A circuit is *delay-insensitive* (DI) if its correct operation is independent of the delays in the logic gates and the interconnections [20] [119]. However, the class of DI circuits has been found to be extremely limited [21] [107]. A restricted form of this class, known as *speed-independent*, allows arbitrary delays in logic elements, but assumes zero delays in the interconnect (i.e. all interconnect wires are equipotential) [41] [124] [125]. Another class of circuits, quite similar to the first two, is known as *quasi delay-insensitive*: i.e. delay insensitivity with isochronic forks (the delays in the arms of a fork are assumed to be the same) which in practice is very close to speed independence [106]. Finally, if the circuit only functions when the delays are below some predefined limit, then the circuit is known as *bounded-delay*. Rather than relying on a bounded delay model of the worst-case path through the circuit, there are a variety of methods for generating a completion signal [150]. Self-timed logic will signal when its output has been composed rather than simply producing a result at some time in the future. These methods use a multiple wire protocol for the communication of data to and from components in a delay-insensitive way. Thus, the circuit's logical behaviour is independent of delays within components and wires. In addition to being freed from the problems of clock distribution, systems designed with these asynchronous circuits are claimed to offer a number of advantages over synchronous designs [66] [106]:

Speed – Asynchronous circuits are optimised for the typical case; worse-case operations simply take longer. There is no fixed clock period during which the operation must complete and therefore delays need only be as long as necessary. This may sometimes be slower than the synchronous clock period, but since the circuits operate at a speed determined by the current operation and therefore are effectively limited by their average (or typical) delay, they are potentially faster. The time variation between worst-case and typical operation can be significant, so optimising a circuit for typical rather than worst-case operations has an advantage not available to the synchronous designer. Generally, these circuits can be smaller and simpler than their synchronous equivalents. Note that the delays themselves are affected by environmental parameters and conditions. Again, synchronous design needs to allow for the worst-case operating conditions to guarantee correct operation.

Power Consumption – Asynchronous circuits generally have a much lower power consumption than their synchronous equivalent. Clocked circuits fire most of their transistors simultaneously at rising or falling clock edges. In asynchronous circuits, since there is no global clock signal, power consumption will be more evenly distributed over time so that the voltage variance should not be as large (transistors only fire when they contribute to the computation). Provided the supply voltage does not fall below the transistor's threshold voltage, an asynchronous circuit would simply slow down but continue to operate correctly [109]. Note that in a synchronous circuit any slowing down could mean the clock transition occurring before data becomes ready, thus causing the circuit to fail.

Also, an asynchronous system activates only those parts of the circuit which are required and so does not dissipate power in the rest of the circuit that is not being used.

Modularity – The complexity caused by the current high level of integration and parallelism makes demands upon our ability to design reliable systems. A key lesson VLSI designers learned from software designers is to divide a problem into modules that can be designed separately.

To reduce complexity, it is necessary for the boundary between modules to be well defined and simple. An important boundary condition is to know when the data communicated are valid. Provided each block in an asynchronous system is internally correct and meets the simple timing constraints of its external interface, the design will be correct in terms of timing. A designer can therefore simply replace one block by another with different characteristics and evaluate any change in performance with little further effort. Again, a synchronous designer does not have this flexibility.

Layout and Robustness – Chip layout is much simplified since the lengths (delays) of the wires do not affect the correctness of the circuit. Similarly delay-insensitive circuits are tolerant to implementation parameters such as fabrication process and transistor scaling.

Metastability – An arbitration device, i.e. a device that grants one of a number requests exclusively, is an example of a circuit exhibiting metastable behaviour. The closer its initial state is to a metastable state, the longer it takes to settle down into a stable state. This problem, first discovered by Chaney and Molnar [28], means that any clocked system containing such a device has a finite probability of malfunctioning.

Automated Synthesis – Accurately estimating the timings of synchronous processors and abstracting them from the logical design is difficult. This has been one of the limiting factors in automatic performance-lead synthesis of synchronous processors. Since the correct operation of an asynchronous circuit is independent of the delays, these circuits have proved attractive for automated synthesis. Many "correct by construction" synthesis methods and compilation tools [19,35,36,66,101,106,116,171] based on the decomposition of formally-proven specifications (e.g. [43]) have been proposed. Due to the complexity of designing asynchronous systems, many recent large designs [37] [110] [170] have been synthesised via compilation tools derived from high-level specifications.

2.4.1 Disadvantages of Asynchronous Design

Asynchronous designs have complexities of their own. First, the logic to detect when data are valid requires extra circuitry. Second, races and hazards need more careful consideration [180]. Output hazards of combinational circuits have little effect on the operation of synchronous systems, as they are allowed to settle before being latched into registers. On the other hand, hazards are intolerable in asynchronous systems because any transition of an output or state variable triggers other transitions immediately; the circuit operates autonomously, and does not depend on any clock timing. For this reason, it is necessary to analyse the circuits used and define the constraints under which no hazard will ever occur [179]. These constraints must then be followed strictly or failure due to hazards may result [119].

Despite the significant work on the specification and design of asynchronous circuits, testing them has received relatively little attention [67] [76]. Traditionally, testing asynchronous circuits has been considered a difficult problem, especially when compared to the synchronous case. Unfortunately, methods used to test synchronous circuits are not directly applicable. This is due, in part, to the absence of the global clock signal in the asynchronous design style which reduces controllability, and makes both the generation of test vectors and the detection of hazards and race conditions harder [22]. However, some techniques have been adapted for use in asynchronous circuits e.g. partial scan path [90] [144]. Other developments have been the inclusion of hazard-free circuit synthesis strategies [179] and fault modelling and fault test evaluation into synthesis systems [145] [173].

2.4.2 Equipotential Regions (revisited)

An equipotential region is one in which a signal can be treated as identical everywhere, that is, the signal requires a negligible amount of time to equalise all potential differences within the designated region. This notion is fundamental in any self-timed methodology [150]. A basic assumption in the synthesis of self-timed modules is that within a module, wire delays are negligible, whereas delays between logic gates are arbitrary but finite. This is equivalent to stipulating that self-timed modules have to reside completely within equipotential regions. In any integrated circuit technology, limits of such regions can be defined, based on the electrical characteristics of interconnects and circuits. Particularly, in MOS technology, equipotential regions are defined within which signals settle in less than τ, the transit time of a transistor [115]. As stated in [150], normally, these limits are much larger than the size of self-timed modules, and hence, no special care is required.

Scaling affects the number of transistors per isochronic region. Suppose that in an isochronic region we allow wires of length at most *l*, with *l* satisfying

$$R.C.l^2 = \beta.\tau$$

for some small constant β. The maximum area of an isochronic region is then $(\beta.\tau)/(R.C)$ and is proportional to $\tau/(R.C)$. Consequently, when scaling down the circuit the maximum area of an isochronic region is divided by α^3. Since scaling multiplies the number of transistors per area by α^2, the maximum number of transistors per isochronic region is divided by α. This implies that subcircuits need to be realised in isochronic regions that are as small as possible and that the minimum number of isochronic regions per chip scales as α^3.

The notion of equipotential regions also brings up another interesting and important point: self-timed modules can be considered to be contained in equipotential regions, communicating with each other reliably through the use of a handshake protocol [150]. Therefore, this protocol must be implemented whenever signals are to be transmitted between regions.

2.4.3 Handshake Protocols

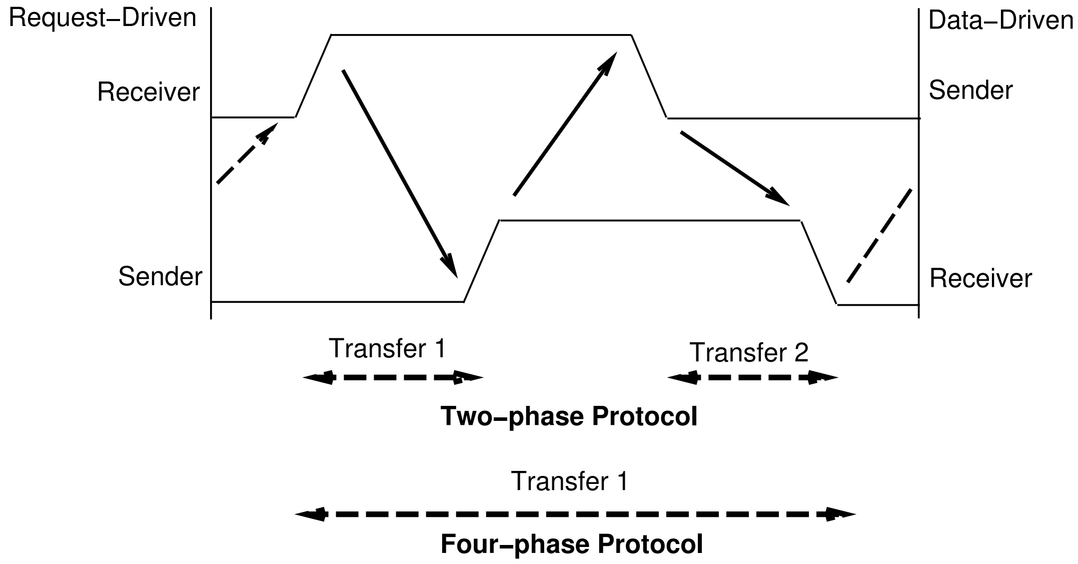


Figure 2–1: Two- and four-phase signalling

A single voltage transition or change of voltage on a wire is the simplest form of signalling that an event has occurred. Since there are time and energy costs associated with changing the voltage on a wire, it pays to use as few voltage transitions as possible in asynchronous signalling conventions, commonly referred to as *handshaking*.

The most efficient signalling convention is *two-phase* handshaking. Consecutive signals or events are indicated by alternating low-to-high and high-to-low voltage transitions. The major advantages of two-phase handshaking, also known as transition signalling or nonreturn-to-zero (NRZ) signalling, are that it is as fast and as energy efficient as possible [150]. However, in practice, additional logic and state information may be required in each element, since logic devices tend to be sensitive to voltage levels or only transitions in a particular direction.

Much of the work on self-timed circuit design has centred around an alternative to two-phase, known as *four-phase* handshaking, which was first used by Muller in many of his examples of speed-independent circuits [117]. In the four-phase handshaking protocol, also referred to as Muller or return-to-zero (RZ) signalling, both wires are initially low, by convention. After each event is sent or presented onto the wire and acknowledged, both wires return to their initial (low) state. The protocol is termed "four phase handshaking" since both transitions (the assertion and the return to zero) are accompanied by additional acknowledgements from the receiver. This results in four phases for a complete message transfer. The principal advantage of this approach is that the nature of four-phase handshaking tends to result in very simple and natural circuit implementations in conventional logic gates. However, it uses twice as many transitions than are necessary and whenever wire delay is a substantial fraction of the operation time, the extra trip required by a single communication can be a serious performance penalty. Figure 2–1 shows both signalling conventions. The terms *request driven* and *data driven* indicate whether it is the receiver or sender who initiate the handshake (the terms *pull* and *push* are also sometimes used).

2.4.4 Data Transmission

The "two-wire" handshake, shown in Figure 2–1, is sufficient to communicate one bit of information to another component. In order to communicate a larger number of bits as a single event, a modification is required to allow the receiver

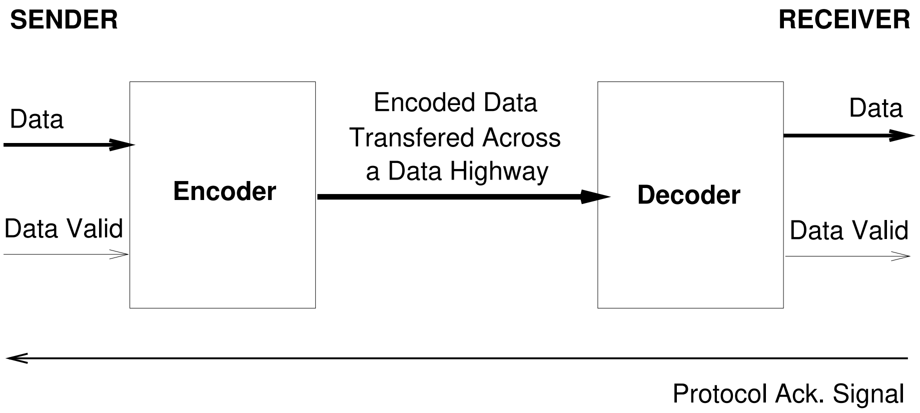


Figure 2–2: Encoded data transmission

to recognise when all the constituent bits are valid. Data transmission can take one of two forms.

Firstly, the data and a data valid signal are encoded together to form a codeword. The transmitted codeword is recognised by the receiver which then extracts the original data (see Figure 2–2). Various codes have been proposed [16] [74] which are dependent on the handshaking convention. The precise conditions for the feasibility of delay-insensitive data communication and a comparison of DI codes has been made by Verhoeff [174]. The most popular one is *Dual-Rail Coding* (DRC) (which is equivalent to Hot codes [74] of length two), because of their simple encoder-decoder pair. In general, the disadvantage of encoded data transfer is the extra circuitry (and therefore, area and performance costs) required to support this mechanism. An encoder and a decoder are required on every output and input data port, respectively. Their area depends on the data width and the coding scheme. Furthermore, the data highway width also depends on the coding scheme, e.g. DRC requires a highway width twice the data width. In practice, for small data widths, dual rail encoding may be quite efficient. But for larger data widths, it becomes expensive in silicon area, in terms of routing the wide data highways across and off-chip, and in terms of the latch sizes associated with holding large code lengths. Although, in the future this may become less of a problem since with scaling, the effective area increases by the square of the scaling factor, and improving technology is increasing the physical area of chips too. Of the other codes suggested [174], Berger Codes seem promising since the data value is a subset of the encoding (i.e. separable), they have a low redundancy, and are easy to code.

An alternative scheme for self-timed datapaths would be to use data path components which operate directly on the DI codeword instead of the data alone. This would remove the need for encoding schemes, (a detection mechanism still being required of course). At first sight, this may seem expensive due to the complexity of the data path components involved, however it has been shown that some designs based on dual-rail encoded data can be comparable in size and speed [108] [131].

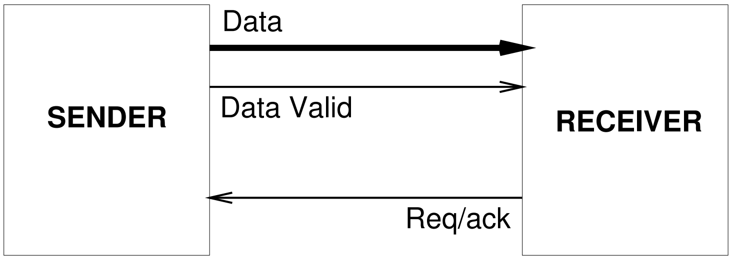


Figure 2–3: Bundled data transfer

The second form is "bundled data transfer" and is based on the bounded delay model. The data wires and the data valid signal are treated as a bundle, i.e. the data valid (DV) signal reaches the receiver after the data wires become valid. This implies that the propagation delay for the data must be less than the delay to propagate the DV signal. In general, this condition is met by inserting an extra delay on the DV wire to account for the worst-case delay on the data wires. This form allows the use of standard datapath components such as multipliers and ALUs without the coding circuitry (as shown in Figure 2–3), thus reducing the communication and area overheads.

The main advantage of this method of building logic functions is that standard techniques or existing cell sets can be used to transform the data and be still used in the framework of a self-timed system. A major disadvantage is that a careful examination of the worst-case delay through the logic block and the data delivery to the receiver is required to guarantee that the bundling constraint is met under all conditions (similar to the task carried out in synchronous designs). Guaranteeing worst-case delay will often require the bundling delay to be large compared to the average case performance of the logic. This not only slows down the module, but also the entire system that uses this module.

Conversions between dual-rail and bounded protocols is simple [150] so that the self-timed logic techniques can be used even in a system that is largely bundled. If dual-rail signalling is used internally on-chip, since dual-rail demands more resources in terms of wires and pins, then it makes sense to convert to a bundled protocol when sending data off-chip.

2.4.5 Ease of Design

In addition to the advantages of asynchronous design outlined earlier, further benefits of reduced design time and costs are also possible. Asynchronous design could be considered easier than synchronous design since the problems with clock distribution, skew and excessive voltage surges may not exist, so a designer need not spend time resolving them. Furthermore, the delays of infrequently used blocks do not significantly effect overall performance, so costly sophisticated design techniques may be avoided. Simpler designs may be used for blocks with data dependent delays (e.g. the ripple-carry adder). The use of high-level design languages derived from CSP [73], such as Tangram [149] and CHP [106] ease the difficulties of designing asynchronous circuits by allowing programs to be automatically compiled to circuits by a silicon compiler [25] [171].

The major drawbacks of self-timed circuits are in the circuit and signalling overheads involved in local communication, and any timing constraints that are required to be met by particular choices of signalling protocols. For example, data may be passed in a delay-insensitive fashion at the expense of using multiple wires per data bit to encode this form of signalling [174]. If bundled data signalling is used instead, the complexity is reduced at the cost of meeting the bundling constraint. Any such timing constraints must be analysed thoroughly and carefully if the circuit is to operate correctly.

2.5 Exploiting Performance

This thesis seeks to exploit the potential performance benefits of asynchrony in processor systems. Care must be taken when comparing synchronous and asynchronous implementations since in practice their design goals are different [2]. One must also be aware of the trade-offs between performance, area and power consumption.

2.5.1 Synchronous versus Asynchronous Control

Events in a synchronous processor are recognised at regular, pre-determined intervals which are ultimately fixed by the clock. If the duration of all actions were constant and known precisely, then the sequencing of actions could be implemented efficiently with a global clock. Unfortunately, the actual delay can vary and is likely to be a lot less than the predetermined worst-case delay, which could result in significant idle periods between events and the next clock tick. In contrast, an asynchronous architecture which is realised by using self-timed components with appropriate handshaking protocols, is able to adjust to varying delays in the components which could be due to data dependencies or changes in the environment. This robustness is at a price, due to the overheads of local handshaking protocols. For this approach to be viable, the average delay of the components together with overheads of self-timed design should be less than the worst-case delays plus overheads of a synchronous design. Synchronisation overheads are difficult to estimate as they are intimately influenced by the clock frequency, technology, fabrication process, routing and chip size.

Most importantly, the self-timed (ST) overhead should not exceed the synchronous overhead by more than the magnitude difference between the average and the worse-case delay of the component. As discussed earlier, while improvements in technology may cause the synchronous overheads to increase, this may not be the case for the overheads due to asynchrony since these can be accounted for by gate delays and local communications. Improvements in performance can be achieved by either reducing the ST overhead directly by speeding up the specific circuits or indirectly by hiding the overhead by doing some “useful work” concurrently. Alternatively, a designer could optimise the design for typical operation. A synchronous designer’s primary goal has been to reduce the worse-case delay (possibly at the cost of increasing the average delay) of components, therefore since the scope for a sufficient margin of difference is small, incorporating synchronously designed components into ST systems may not prove advantageous. Furthermore, when components are connected in pipelines or arrays, the overall performance will tend towards the worse-case value since throughput is limited by the slowest individual component stage [87] [97]. Consequently, in comparison to an equivalent synchronous design, the performance may even be worse due to the ST overheads. Previous attempts to harness this proposed advantage of self-timed circuits have not proved too successful [146] [156].

2.6 Pipelines

Pipelining is an implementation technique whereby a cascade of processing stages is connected (generally in a linear fashion) to perform functions over a stream of data flowing through the stages. This technique, which is by far the most popular method for enhancing performance in CPUs, provides a way to start a new task before an old one has been completed.

The throughput of a pipeline is determined by how often a result exits the pipeline. In a synchronous pipeline all of the stages must be ready to proceed at the same time. The time required to move data down one stage of the pipeline, the machine cycle, is determined by the time required by the slowest pipe stage. As long as there are no dependencies between the data, the throughput is fixed at one result per machine cycle. Data flow between adjacent stages in an asynchronous pipeline is controlled by a handshaking protocol. Results only move forward when the succeeding stage is empty. An asynchronous pipeline may have a variable throughput rate since different stages may experience different delays. For complex (data-dependent) computations, asynchronous design has the advantage of exploiting the actual delays, whereas synchronous solutions are adjusted to the worst-case.

2.6.1 The Conversion of Synchronous Pipelines to Equivalent Asynchronous Ones

This section describes the transformation of a synchronous pipeline to an equivalent asynchronous one, as illustrated in Figure 2–4. Part (a) illustrates a conventional synchronous pipeline with a clock signal being used to control the transfer of data between functional units (FUs), and by the control unit (CU), to generate the correct sequence of control signals to define the pipeline's

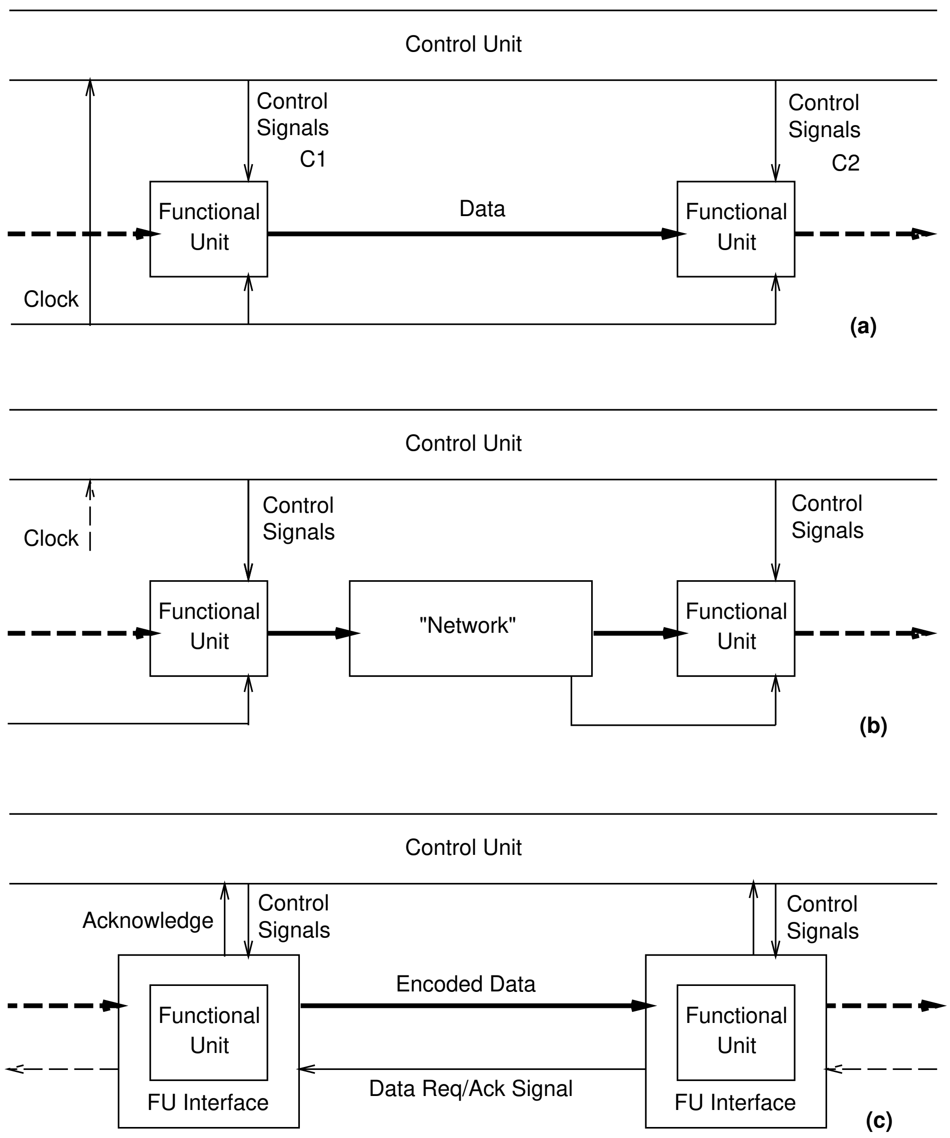


Figure 2–4: From a synchronous to an asynchronous pipeline

operation. In the CU, the relationship between control signals *C1* and *C2* is strictly bound since they must be generated at the correct time and in correct order. In other words, the CU needs to incorporate a (pessimistic) "timing" model of the pipeline. A simpler pipeline as in the case of RISC architectures, results in simpler control and therefore smaller control costs.

Part (b) illustrates an intermediate stage, where the transfer of data is controlled locally. The *"network"* is responsible for communicating data and control signals between FUs. This process can be as simple as bundled data transfer [158], or more complex such as encoding the data prior to transfer and decoding it at the receiver [174]. The clock (to the CU) is now only used as a time reference for the generation of FU control signals. The CU still needs to model the timing characteristics of the pipeline which results in minimal performance gains, if any. However, the global clock signal has been removed by the decentralisation of communication controls.

Part(c) illustrates a truly self-timed pipeline. The interfaces receive control signals from the CU and encode transfer data for detection at the interface of the destination. When valid data has been detected and latched, the interface sends an acknowledgement signal back to sender. It is now able to remove the data and release the bus (if shared). The interface is responsible for meeting the operational requirements of the FU, such as guaranteeing that the input data is valid before control signals are asserted. This, together with the communication protocol, decouples the logical behaviour from the timing characteristics of the pipeline. This enables functionally-equivalent FUs to be interchanged without affecting the operation of the rest of the pipeline. Since the CU no longer requires the timing characteristics, the pipeline control becomes less complex and therefore faster. The control signals C1 and C2 no longer have to be generated at the right time or in the correct order with respect to each other, since a FU cannot begin its operation until it has received both the data and the control signals (due to the FU interface). The only constraint on the control or data signals is that the previous value must have been received by the corresponding FU interface before the next one can be issued. This means that both the CU and the interfaces cannot change the value of a signal until it has received an acknowledgement from the receiver. A typical handshake cycle might be as follows: wait until FU is not busy; assert the control signals; wait for an acknowledgement; clear control signals; repeat. This naturally maps to a four-phase protocol [150] with the acknowledgement signal also doubling as a busy flag.

This would allow the control unit of a processor to use the acknowledgement signals from FUs as part of a scoreboarding mechanism.

The CU cannot predict exactly when the FU with the largest delay in the pipeline will finish. By letting the FU indicate that it has finished, and not necessarily to the control unit but to its successor, the pipeline is driven by local average delays and not centrally-fixed worst case ones.

2.6.2 Micropipelines

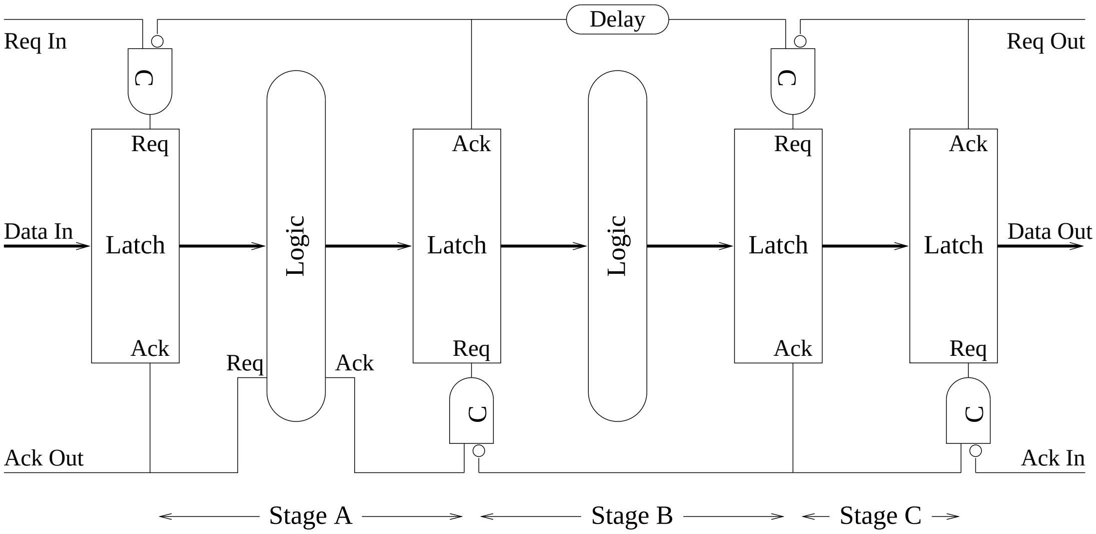


Figure 2–5: A basic micropipeline FIFO

In the 1988 Turing Award lecture, Ivan Sutherland outlined a methodology for the design of asynchronous pipelined systems using the two-phase bounded-delay (bundled data transfer) protocol [158]. The interface has an arbitrary number of data bits accompanied by two signalling wires (req and ack). A micropipeline is a simple event-driven elastic pipeline which maintains the order of data. A block diagram of a generic micropipeline is shown in Figure 2–5. It consists of three parts: a control network consisting of a single C-element per micropipeline stage, a latch in each stage, and possibly some combinational logic between stages. The logic can signal its own completion (Stage A), or it can be simulated with a known delay (Stage B). If no processing is present between stages, the pipeline becomes a first-in first-out (FIFO) queue (Stage C).

2.7 Related Work

Udding [165] [166] first proposed a formal definition and classification of delay-insensitive circuits. Since then much theoretical work has evolved from process algebra [82], trace theory [41] [140] [141] [172] and Petri nets [31] [101] [116] [178]. Due to the complexity of designing asynchronous systems, many large designs have been synthesised via compilation tools derived from high-level methods. These circuits have been shown to be efficient and robust in the design of control circuitry [36] [74] [106] [177]. At the board level, communication interfaces such as the VME protocol [68] already make use of asynchrony [99]. However, these circuits have been considered inadequate for designing data paths for the following reasons. The overhead of encoding data, generating completion signals and arbitration on buses make them slow and wasteful in area [2] [135]. Nevertheless, a few fully asynchronous microprocessors have been proposed. Many of these designs have concentrated on specific aspects of self-timing such as their formal synthesis [37], low power consumption [48], or just the feasibility of implementing conventional microprocessor architectures (with little emphasis on their performance or efficient operation).

The first asynchronous VLSI processor was built by Martin [110] at California Institute of Technology. The goal was to demonstrate that complex circuits could be generated from specifications using a library of self-timed elements. The Amulet project [56] [137] at Manchester University investigated the application of asynchronous micropipeline techniques to the commercial low-power ARM microprocessor. The NSR processor [18] built at University of Utah is a general purpose processor built from Actel FPGAs. In addition to being internally self-timed, the units are decoupled through self-timed FIFO queues between each of the units which allows a high degree of overlap in instruction execution. Other processors which are still in their design stages (or have yet to be built) include: SCALP [49] and Hades [47], which are superscalar designs; TITAC, which is a simple 8-bit processor built using CMOS gate array technology [129]; the ECSTAC [123] processor which uses an 8-bit architecture and a two-phase communication strategy; and STRiP which, although it is called "self-timed", is in fact a synchronous processor which can dynamically alter its clock period [39].

Although these designs are based on a single micropipeline-style datapath [93] [158], viewing the datapath as a linear sequence of stages may not be very efficient for reasons elaborated in the following section. A couple of designs have begun to investigate the influence self-timing has on processor architectures. A novel architecture has been recently proposed by Sproull *et al.* at SUN Microsystems called the Counterflow Pipeline Processor Architecture [157], which derives its name from the fact that instructions and results flow in opposite directions in a pipeline and interact as they pass (similar to a 1-D systolic array). It supports a form of register renaming, data forwarding, and speculative execution across control flow changes. The performance of such an architecture is still unknown [152]. Fred [142] is a decoupled, pipelined architecture which supports dynamic instruction re-ordering and out-of-order instruction completion.

2.8 This Thesis

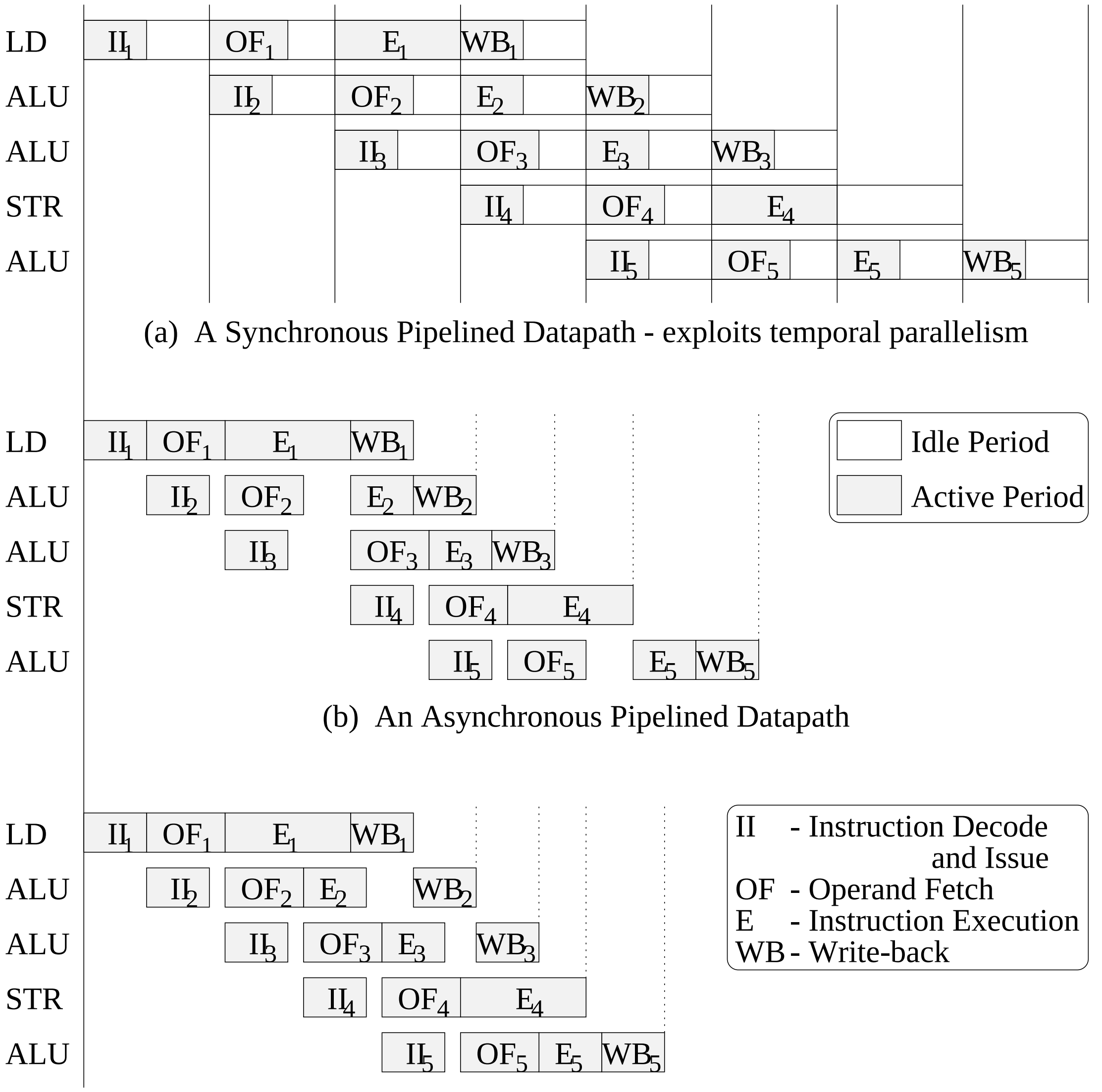
One feature common to all of these processor designs is their view of the datapath. As with synchronous designs, the datapath is still viewed as a single linear pipeline. The work described in this thesis differs from them by viewing a datapath as a network of asynchronously communicating resources through the generalisation of the micropipeline concept to a network of communicating pipelines.

2.8.1 Towards Asynchronous Datapaths

The clock period of a synchronous pipeline is determined by the delay of the slowest stage which takes into account worst-case timings for execution and propagation. Furthermore, optimal performance for a pipeline is achieved when all the stages are balanced. This is quite difficult to achieve in practice, since the stages of a typical pipeline perform different operations, and often their delays are data-dependent. Figure 2–6(a) illustrates the operation of such a datapath in which synchronisation overheads have been omitted for the sake of clarity. This imbalance between stage delays results in idle periods leading to poor utilisation of the physical resources. Of course, further pipelining of the slower stages could reduce this at the cost of increased design complexity and synchronisation overheads.

In contrast, the performance of an asynchronous pipeline is determined by the actual delays of individual stages (usually the average delays), and overheads due to self-timing protocols (which have been omitted in Figure 2–6(b), again for the sake of clarity). ([54] compares synchronous and asynchronous pipelines, taking into account their overheads.) This pipeline only exploits temporal parallelism as before, but does so more efficiently. An instruction proceeds to the next stage once it has completed the current one and the next stage is free. Although each stage may consist of a number of (different) resources, generally, only one (or a subset) of them will be active at any time for a given instruction.

The average throughput of any asynchronous pipeline cannot be greater than the average throughput of the stage with the slowest isolated average performance [128]. This is only the upper bound and thus may not always be



(a) A Synchronous Pipelined Datapath - exploits temporal parallelism

(b) An Asynchronous Pipelined Datapath

(c) An Asynchronous "Networked" Datapath - exploits spatial parallelism as well

Figure 2–6: Synchronous and asynchronous pipelines

achieved, especially since once a stage is idle it is no longer able to maintain its isolated average performance. Idle times, caused by blocking and starvation, can be reduced by introducing additional buffers between stages (the number required being closely correlated to the coefficient of variation of data dependent delays between the stages [87]). However, this increases pipeline latency and area costs, possibly resulting in reduced area-time performance and therefore comparing unfavourably with a synchronous equivalent. Exploiting spatial parallelism, through the improved utilisation of resources, not only reduces idle times but may also reduce the number of buffers required to maintain isolated average performances. Thus, an implementation technique which is more flexible than a linear pipeline is required to model datapath behaviour efficiently.

Figure 2–6(c) illustrates an asynchronous datapath which exploits spatial parallelism within some of the stages. (The datapath is no longer modelled as a true pipeline). Successive instructions which utilise different resources within a stage are now able to execute concurrently. In the simple example under consideration in Figure 2–6(c), the execute stage has two concurrent resources. It is possible for the instructions to share resources in any of the stages. For example, while an instruction is stalled waiting for an operand on one bus, another instruction could use the other buses to fetch its operands. The amount of spatial parallelism which can be exposed in practice is determined by the relative delays of the functional units in the datapath.

2.9 Micronets

Micropipelines [158] have been used to model linear asynchronous pipelines such as datapaths [56] [143], and two-dimensional pipeline structures [64]. However, as described earlier, viewing a datapath as a single linear pipeline does have limitations. A new paradigm called *micronets* is proposed for the distribution of control in asynchronous processor architectures. Micronets model datapaths as a network of communicating functional units which allows the efficient exploitation of both fine-grained instruction-level parallelism and the actual execution costs of instructions.

2.9.1 Micronets, Microagents and their Micro-operations

In a synchronous datapath the centralised control forces each instruction to go through all the stages regardless of its need to do so (in effect a single pipeline). The cost of execution is determined by the worst-case estimate of the slowest stage. The same is true of a micropipeline-based datapath [56], except that the cost is now determined by the actual delay of the slowest stage.

Micronets are effectively a generalisation of Sutherland's micropipelines. The components within each of the micropipeline datapath stages are exposed in the form of fine-grain *microagents*. The microagents in any "stage" can operate concurrently, and microagents in the different "stages" communicate with each other asynchronously. Program instructions only utilise the relevant microagents and for just as long as is necessary. More than one instruction may utilise the different microagents within a "stage". Figure 2–7 compares the resource utilisation in micropipelined and micronet datapaths. In the former, the number of active instructions is never greater than the number of pipeline stages, and at any time only a subset of the resources in each of the stages is normally utilised. In micronets, the number of instructions which may be active at any time is bounded by the number of microagents. An instruction which does not require any of the resources within a "stage" can skip it. Furthermore, the time spent by instructions in microagents may vary. Due to these reasons computations may overtake. In this way, micronets differ from 2-D micropipelines [64] which represent asynchronous regular arrays. This feature will be exploited to implement out-of-order instruction completion. (Note also that a microagent itself can consist of a number of (micro)pipeline stages).

Figure 2–7(b) shows an instruction (I1) executing concurrently with a succeeding instruction (I2) in what would have been the same stage in a synchronous pipeline. Because there are effectively a number of paths, different instructions need not necessarily complete in the order they were initiated. Also,

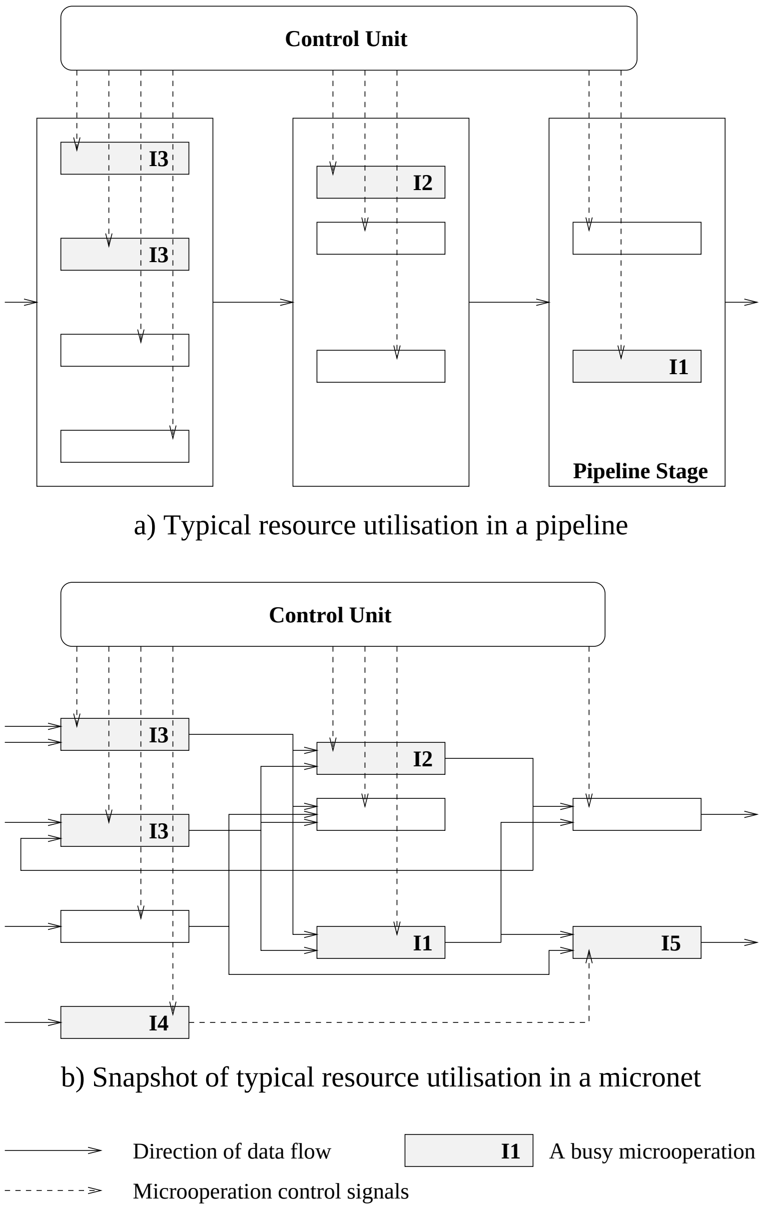


Figure 2–7: Contrasting a micropipeline with a micronet

the micronet is controlled at two levels: the data transfer between the micro-agents is controlled locally, whereas the choice of micro-operations within the microagent and the destinations of the results are controlled by the control unit or by other microagents (see I4 and I5 in the figure). Communication between microagents may occur either across dedicated lines or via shared buses. The micro-operation control signals can also be used to prevent contention on shared buses. There are no specific restrictions on the choice of handshake protocol employed at the different control levels. However in practice, such a choice is influenced by performance and area considerations.

2.9.2 Micronet-based Datapaths

The micronet control paradigm is investigated in the context of a Reduced Instruction Set (RISC) architecture. Self-timed circuits are used to distribute processor control away from a centralised Control Unit (CU) (found in conventional synchronous processors) to autonomous functional units. This distribution of control locally to functional modules affords greater scope for exploiting concurrency between instructions.

Data dependencies within synchronous datapaths are resolved by using either a hardware or a software interlock [70], which adds to their control complexity. A micronet datapath uses existing handshaking mechanisms together with simple locking of registers to achieve the same effect with trivial hardware overheads. In synchronous designs the structural hazards are normally avoided in hardware by using a scoreboarding mechanism. In micronets this is provided by existing handshaking protocols. The choice of a four-phase communication protocol [150] between the functional units allows the efficient utilisation of these resources, by avoiding the additional control costs (scoreboarding and hazard avoidance mechanisms) normally associated with processors which exploit ILP. (This choice and its justification is discussed in greater detail in Chapter 5). Out-of-order instruction completion can be supported in synchronous designs, but at a non-trivial cost. Micronets are able to relax the strict ordering of instruction completions and thereby exploit further ILP. A Micronet-based Asynchronous Processor (MAP) design has the advantage of exploiting the best-case delay (behaviour), whereas synchronous solutions are adjusted *a priori* to the worst-case. The result is an increase the utilisation of the functional units by reducing their stalls. By exploiting both ILP and actual run-times of instructions, better program performances may be achievable by asynchronous processors.

2.10 Summary

There has been renewed interest in asynchronous circuits, especially in a restricted form known as self-timed circuits [150]. These circuits have a number of advantages [106], including their automatic synthesis from specifications [66]. While this has resulted in provably-correct circuit designs, the performance of the resulting processor architectures have been largely overlooked.

A few processors have been proposed [56] [123] [143] which utilise asynchrony at the circuit level and exploit average-case behaviour for better performance. However, in the only comparison of an asynchronous processor with its synchronous equivalent, results showed the synchronous version to be faster, smaller and at the same time consume less power [56]. One reason could be that the chosen architecture itself is better suited to a synchronous control paradigm. This is emphasised by the fact that the next design will include architectural modifications [55] (rendering a comparison to the original synchronous version unfair). This underlines the fact that the design of a processor must consider the relationship between different aspects of the system.

A new model has been proposed called the *micronet* for modelling asynchronous datapaths, which efficiently exploits actual instruction execution times and instruction-level parallelism. Micronets model processor architectures as a network of communicating resources, in contrast to the traditional one of a linear pipeline. Micronets distribute the control to the functional units, which enables the exploitation of fine-grain concurrency between instructions. It will be shown that the overheads due to asynchrony can be hidden with the four-phase protocol being used to implement scoreboarding and hazard avoidance mechanisms, without incurring additional control costs. Although improvements may be obtained in one area of the system design, it is imperative that this is not at the expense of performance in another, thus having an overall negative effect on the system. Therefore, the following chapters examine the influence of this novel asynchronous control paradigm on the design of processor architectures. In particular, the instruction latencies and resource utilisation in a micronet architecture will be investigated together with the compiler's ability to schedule code for this target.

Chapter 3

A Parallel Event-Driven Simulator

> *"Both users and designers of computer systems are interested in performance evaluation since their goal is to obtain or provide the highest performance at the lowest cost."* [80]

3.1 Introduction

The dynamic behaviour of asynchronous systems is difficult to model analytically for making accurate performance predictions. The approach adopted in this work has been to simulate register-transfer-level (RTL) models augmented with timing obtained from SPICE-simulations of their circuit implementations. This chapter describes the development and implementation of an asynchronous parallel event-driven simulation platform for the performance evaluation of both synchronous and asynchronous processor architectures and systems. One objective was to develop a simulator for obtaining performance figures for the execution of algorithms under different scheduling or placement strategies, on different (multi)processor architectures and interconnection topologies. In particular, this would include the measurement of the performance over time of an ensemble of heterogeneous functional units which operate concurrently and communicate with each other asynchronously. This tool is the workbench for the work described in this thesis.

3.2 Parallel Discrete Event-driven Simulation

Logic simulation is a common and effective technique for investigating the behaviour of computer designs. However, accurate simulations of large designs can be extremely time-consuming. By executing them on parallel architectures, Parallel Discrete Event-driven Simulation (PDES) attempts to address this problem by exploiting the structural concurrency inherent in the applications.

A Parallel Event-driven Processor Simulation Environment (PEPSÉ – pronounced in the same way as the well known fizzy drink) has been developed based on the ELSA algorithm [8]. PEPSÉ provides a framework for efficiently evaluating the performances of both sequential and parallel architectures. The architectural components may be modelled either uniformly at one of the different levels of abstraction, or the components can be modelled individually at different levels. One could for example examine the performances of cache coherence protocols in shared memory MIMD machines, communication protocols for local area network, effects of topology in distributed memory MIMD machines, resource hot spots within processor design, to name just a few. For our purposes, architectures are modelled at the register-transfer level with accurate timing delays of the functional units being provided by SPICE simulations of their VLSI implementation.

The current implementation of PEPSÉ runs on a network of transputers called the ME*i*KO Computing Surface [79]. The architectures are modelled in the programming language Occam2 [78]. (Occam has long been used to specify the behaviour of circuits [103] [105].) A system can be described as a collection of concurrent processes which communicate with each other asynchronously through channels. The semantics of Occam2 captures the behaviour of asynchronous circuits naturally [161]. The asynchronous nature of the underlying simulation algorithm efficiently simulates the class of architectures under investigation (compared to time-driven simulations). For typical sizes of system-under-simulation (s-u-s), these runs could be in the order of a few hours on a uniprocessor. PEPSÉ exploits the structural concurrency in the s-u-s to reduce these run times considerably.

3.3 An Overview of PEPSÉ

The simulator is comprised of a number of components, as shown in Figure 3–1.

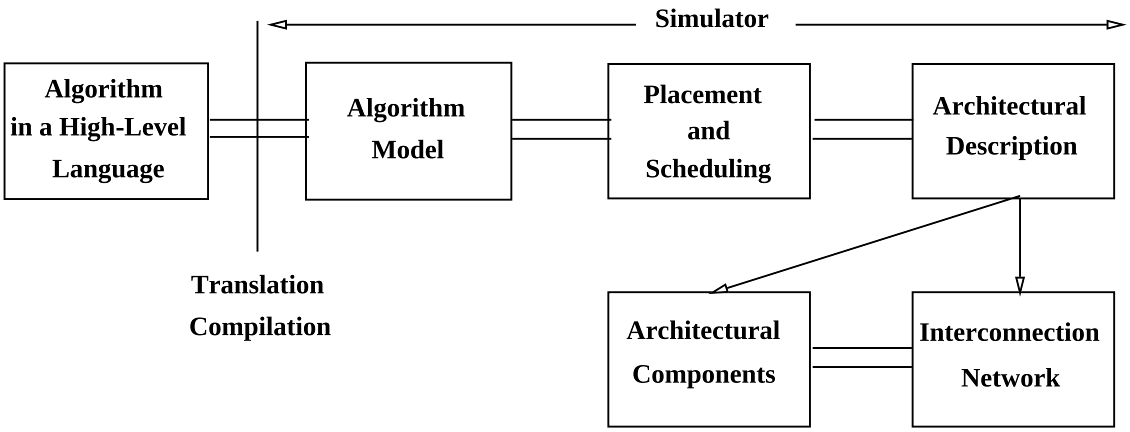


Figure 3–1: Overview of the simulator

Algorithm in HLL – This is the application program/software which is to be executed on the simulated architecture. The application program is usually in the format of a high level language, and will need to be "compiled" into a format which is suitable for the particular architecture upon which it is to be executed. This "compiled" format is called the Architecture Specific Code.

Algorithm Model – This is the Architecture Specific Code (ASC) of the application program. The ASC contains instructions specific to the processor or architecture upon which the application is to be simulated. Whether the ASC is equivalent to assembler, machine code or some other intermediate code depends on the level at which the processor is modelled. For example, for register-transfer level models, the ASC would normally be in the form of assembler instructions from the processor's instruction set. Since we are interested in the performance of an algorithm on a given architecture, it would also be necessary to take account of compiler characteristics.

Placement and Scheduling – This is the strategy for distributing the ASC over the processor architecture, and determining how it is to be scheduled. (Currently this task is achieved manually.)

Architectural Description – This consists of two groups:

1. The Architectural Components which include: processors (which consist of two objects, an instruction fetch object and an instruction execute object, for modelling SIMD architectures or instruction prefetch mechanisms), synchronous processors with clock speed as a parameter; memories or caches whose parameters include size, access time and initial contents; and application specific hardware which includes components from logic gates to application-specific integrated circuits (ASICs).
2. The Interconnection Network which describes the communication between the architectural components. Direct or point-to-point connections between two objects to model simplex communication can be achieved using the Occam2 communication channels. Shared connections, such as a bus, need to be modelled by a simulation object. These objects have both a propagation delay and the number of

components which share the bus as parameters. Half duplex communication can be modelled as a bus with two ports, and full duplex communication as two simplex ones.

3.3.1 The Simulation Platform

The simulation platform is based on ELSA algorithm. In ELSA, logical processes have their own local simulation clock and communicate with other processes via time-stamped (duration bounded) messages. Each logical process or *simulation object* consists of two components, firstly a behavioural model of the object which evaluates the physical process' operation based on the value of its inputs at the current simulation time and secondly, a mechanism to control the local simulation clock and time-stamping of output messages. This mechanism uses the delay associated with the particular operation to generate the correct time-stamped output. The simulation proceeds asynchronously, with each logical process passing state information in the form of tuples via their simulation platform, as shown in Figure 3–2.

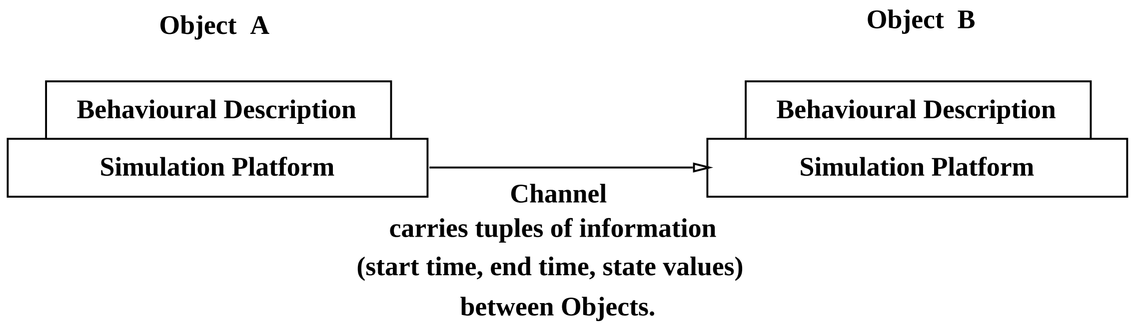


Figure 3–2: The simulation platform.

Each tuple of information contains:

1. a set of *state values*, and

2. a *start time* and an *end time* which defines the interval for which these state values are valid.

Note that a tuple containing a start time equal to the end time conveys no useful information and that all tuples on each channel must represent contiguous periods of time.

3.3.2 The Basic Simulation Platform Algorithm

The following steps outline the basic simulation platform algorithm:

Algorithm 1 : ***Basic Simulation Platform***

1 Initialisation of variables and flags.

2 Clear input buffers, set *input start* and *end times* = 0,
  and place initial output values in output buffers.

3 Send the initial output tuples out on their respective output channels.
  Set the object's *current simulated time* = 0.

4 If necessary, get required tuples from each input channel.
  while *(current simulation time* $\geq$ tuple's *end time)* get the next tuple.

5 Evaluate the function (output states values)
  based on current inputs using the behavioural description.

6 Calculate the start time of all of the output tuples.
  start time = *current simulation time* + *object delay time* δ

7 Calculate the end time of all of the output tuples.
  end time of each output tuple = MIN(*end time* of all input tuples) + δ

8 Send the output tuples which are still within the simulation window,
  i.e. tuples which have *start time* < "Stop Simulating" time.

9 Update simulation time.
  current simulation time = MIN(*end time*) of all the inputs tuples

10 if not finished simulating, i.e. still within the simulation window,
  then goto step 4.

11 Sink outstanding tuples, i.e. those tuples which have *start times* that
  are outside the simulation window.

Steps **1** and **2** are initialisation stages, with Step **3** sending the initial tuples defining the output states for the period (0, δ) at the start of the simulation. The only inputs which can affect the state values during the period (0, δ) are those with *start times* < 0, which obviously do not exist. Steps **4** to **10** constitute the main loop where each time a new tuple(s) is required to advance the object simulation time, a re-evaluation of the output states takes place. Step **6** evaluates the output *start time* which is the how far into the simulation the object has progressed plus δ, a delay for the generation of the output state values. Step **7** determines the output *end times* which are set to the time at which the next "event" occurs, which will be at the earliest *end time* of all of the input tuples, plus δ the same delay for the generation of output state values. At Step **9**, the current simulation time of the object is advanced to the time at which the next "event" occurs. This means a new tuple(s) will be required and therefore a re-evaluation of the output values. Step **11** is more an implementation requirement to guarantee that all objects will complete executing and terminate.

The propagation delays over dedicated wires (one-to-one connections) are modelled by incorporating them directly into the source object, and delays on shared wires are modelled as a separate resources. If necessary, the simulation platform for these resources can easily be made to detect instances of contention.

3.3.3 The Class Models

Using the basic simulation platform together with its behavioural description is sufficient to allow the simulation of an object, if the output state(s) of that object are a function of only the current input(s), as in the case of simple logic gates:

$$\textit{output states(time} + \delta\textit{)} = \textit{f(input states(time))}$$

Since the simulation occurs at the instruction/register transfer level, most objects have more complex behaviours such as state machines. This means that the output states are a function of both the input states and some internal state of the object:

$$output\ state(time + \delta) = f(input\ states(time) + internal\ state(time))$$

This means it is necessary to modify the basic simulation platform. Another reason for modifying the simulation platform of some objects is related to performance. In order to achieve good performance on parallel systems it is necessary to keep inter-processor communication to minimum.

Clocked Objects

State machines, registers, synchronous processors etc., all require some sort of "clock" or latch signal. These objects are generally only sensitive to the value of input signals at the transition of (or when a certain value occurs) on one of the inputs, i.e. the clock. If an object has a clock input then the simulation platform need only evaluate the outputs once every clock period, instead of each time the object needs a new tuple. In practice, these clock/latch signals can either be:

- regular/periodic or irregular/aperiodic, and either
- edge- or level-triggered.

For periodic clock signals, the simulation platform will know when the clock transitions will occur. For example, if the clock input signal is regular, e.g. from an oscillator, the clock input signal can be modelled internally within the object. However for aperiodic clock signals, the simulation platform will have to test only the state value of the clock input to determine its timing information. An alternative would be to wait for a transition on the clock input and then allow the behavioural description to test the clock input along with the other inputs when evaluating the outputs. Remember, even if outputs do not change it will still be necessary to send new output tuples to allow the simulation to proceed. The effect of the clocked inputs on the basic simulation platform is discussed in the following sections.

Objects with Irregular Clock Signals

This simulation platform need only evaluate the outputs when there is a new tuple on the clock input, therefore the platform only considers the tuples on the clock input as new events. This implies that the simulation only uses the clock input for the generation of timing information. Each iteration of the simulation loop will require a new clock input tuple with the corresponding tuples of the other inputs being required to evaluate the output tuples.

The Simulation Algorithm

The steps of the basic algorithm requiring modification are:

4. Get the required tuples.

- On the clock input:
  - if (*current simulation time* == *end time*) then get the next tuple.
- For each of the others:
  - while (*current simulation time* $\geq$ *end time*) get the next tuple.

7. Calculate the end time of all of the output tuples:

- *end time* = clock input's *end time* + *object delay time* δ.

9. Update simulation time:

- *current simulation time* = clock input's *end time*.

Objects with Regular Clock Signals

The simulation platform for an object of this type is a special case of the one with an irregular clock signal. The simulation advances a fixed (and known) amount of time, i.e. the clock period, each iteration and therefore there is no need for a separate clock input. Even if no new input tuples are required or the input states do not change over a number of iterations, it is still necessary to re-evaluate the outputs since the timing information will need to be updated, and being a clocked object, the outputs are likely to be functions of both the inputs and the internal state of the component.

The Simulation Algorithm

The variable *object latency* can be used as an offset or time delay before the periodic clock starts.

3. Send initial tuples:

- *current simulation time* = *object latency*.
- Each input tuple's *start time* = *current simulation time*.

4. For each input, make sure the tuple is valid:

- while (*current simulation time* $\geq$ *end time*) get the next tuple.

7. Calculate the end times of all output tuples:

- *end time* = *current simulation time* + *object delay time* δ + clock period.

9. Update simulation time:

- *current simulation time* = *current simulation time* + clock period.

Level-triggered Clock Signals

A level-triggered clock input is treated just as another input since this input will only have a boolean effect on the other inputs. Therefore, the basic simulation platform would suffice. However, employing the simulation algorithm used for irregular clocked objects may generate fewer output tuples.

3.4 Development Notes

The PEPSÉ simulation platform was implemented in Occam2 [78]. Occam2 supports concurrent threads of execution (processes) and uses unbuffered channels to provide synchronisation and communication between processes. However, since synchronisation is not required, the channels are buffered to avoid deadlock, decouple logical processes (thus increasing concurrency), and reduce message traffic by merging tuples.

3.4.1 Occam Buffers

Avoiding Deadlock

Deadlock will occur if a cyclic relationship exists between a group of objects. Initially, all of the objects will attempt to place their initial tuple on their output channels, including the recipient objects, and will therefore be unable to receive the incoming tuple. Buffers have been inserted to overcome this communication constraint within Occam2. These buffers simply receive tuples, releasing the sending object, and pass them on to the receiving object when it is ready, thus having the effect of unblocking the objects not only at their initial transmission but also at any time two objects attempt to send tuples simultaneously to each other.

Maintaining Asynchrony

The use of buffers also allows objects which can proceed "quicker" into the simulation not to be held up by "slower" ones. Buffers can queue tuples, thus allowing the sender object to proceed, by removing the synchronisation between sending and receiving objects. However, one factor which has been observed while simulating at the register/instruction level is that, in general, there is a tight cyclic relationship between some pairs of objects, especially self-timed components. If object A has an output channel to object B, object B is quite likely to have an output channel to object A. In this case, there seems to be only one outstanding tuple in the queue, and this occurs since both objects are progressing at about the same rate.

Aliasing Variable Names

In order to generalise the simulation platform, since the number of inputs and outputs to a particular object varies, the simulation platform takes an array of inputs and an array of outputs. The buffers allow the aliasing of these array variable names from the output of one object to the input of another object.

Numbers of Tuples on a Channel

With the basic simulation platform, the total number of tuples on each output channel (one tuple per channel per iteration of the simulation loop) can be bounded below by the largest number of tuples on any of the inputs and bounded above by the sum of all the tuples on each of the inputs.

With regular clocked objects the number of tuples on each output will be the simulation duration divided by the clock period. Also, with irregular clocked objects, the number of tuples on each output will be equal to the number of tuples on the clock input. Thus, clocked objects prevent the avalanching effect on tuple numbers. Furthermore, the buffers can be used to merge consecutive tuples with identical state values and thereby reduce message traffic.

3.4.2 Guarded Outputs

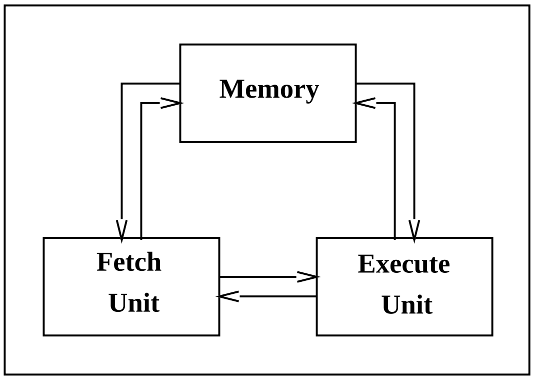


Figure 3–3: A microprocessor model

In some cases, it is not necessary to send tuples on all of the outputs at every iteration. For example, when the fetch unit passes a load instruction to the execute unit, it will take one iteration of the execute unit to interpret the instruction and initiate a read from memory, and one further iteration to execute the instruction. This implies that, after the first iteration, the execute unit object sends a tuple to:

- memory, a read access request,
- the instruction processor, a null tuple or unexecuted instruction message.

After the second iteration, the object sends a tuple to:

- memory, a null or no access request message,
- the instruction processor, the updated register values.

In order to execute one instruction, the execute unit object had to send two tuples on each of its output channels, of which only one conveyed useful information. Guarded outputs are boolean flags which inhibit or allow the transmission of a tuple on a particular output channel. By applying guarded outputs to the above example, the data processor object would not send a tuple to the instruction processor after the first iteration, and to the memory after the second.

Thus, the use of guarded outputs can achieve a significant reduction in the number of tuples used, without losing the modularity between timing information generated in the simulation platform and the state information generated by the behavioural description.

3.4.3 Modelling Signals

The transfer of state information takes place via tuples which are represented as a variable length array of integers. Each tuple has a number of flags associated it, these being index values within the array:

elsa.tup.len – is the pointer to the tuple length (index value "0"),

elsa.start.time – is the pointer to the time from when the states are valid (index value "1"),

elsa.end.time – is the pointer to the time until when the states are valid (index value "2"),

elsa.state – is the pointer to the first state value (index value "3"). The number of states within a tuple can be determined by

number of states = tuple[elsa.tup.len] - 3

The use of variable length arrays allows the ability to incorporate a number of states into one tuple and thus reduce the number of communication channels between two objects in any one direction to 1. This will always be true unless the start- and end-times of particular states need to be different, in which case another channel and separate tuples would be required.

3.5 Component Delays

The fidelity of the simulation results is determined by the accuracy of the simulation model. The models used in this thesis have been validated via a combination of HSPICE simulations and analytical analysis. Each of the architectural components used in the designs of Chapter 4 has been modelled as an individual simulation object based on a 1.2 μm CMOS process implementation of off-the-shelf/standard library components. Individual component delays have been extracted from a simulation tool within ES2's commercially available silicon compilation integrated circuit design suite SOLO 1400 [50]. (ES2 claim to guarantee circuits designed using these tools will be fully functional on first silicon). In the synchronous design, component delays were based on worst-case timings, (including component operation e.g. propagating a carry the entire length of the adder's carry chain) and nominal/typical timing delays (average component operation delays) in the self-timed case. Unfortunately these designs were not laid out completely since this tool was not suited to custom datapath design and thus full account of propagation delays were not considered. The designs described in Chapter 5 were based on the EUROCHIP 0.7 μm CMOS process obtained from the CADENCE design suite. These tools are better suited to datapath design (giving the designer more control over layout) thus the HSPICE simulations give a more accurate account of both relative component and propagation delays.

3.6 Conclusions

PEPSÉ provides an efficient framework for obtaining accurate performance figures for the execution of small programs on the simulated architectures. By allowing mixed-level simulations the run-time costs can be further reduced without sacrificing accuracy.

The approach adopted here is well suited to the simulation of asynchronous circuits due to the asynchronous nature of the underlying algorithm itself. This algorithm is inherently deadlock free and, in its conservative form, never violates the causality principle which means that an expensive roll-back mechanism is not required [8].

Chapter 4

The Control Paradigm and the Instruction Set

4.1 Introduction

In general, improvements in the performance of processor architectures can be achieved in two ways: reducing the time taken to complete a unit of work (i.e., reduce the latency of the operation) or by increasing the amount of work achieved per unit time (i.e., increase the concurrency between operations). This chapter focuses on the former by comparing an *asynchronous* control paradigm, where the datapath control is distributed and functional blocks communicate using handshaking protocols, with the traditional synchronous style. Specifically, this work attempts to investigate if any performance improvements in the execution times of individual instructions can be obtained within a typical RISC datapath implemented as a micronet.

Although asynchronous control of datapaths had previously been considered too expensive [2] [135], other work has suggested that the opportunity for improved performance does exist [38] [63]. This chapter investigates the application of the asynchronous control paradigm to a variable length pipelined datapath and compares the effect of the two design styles, synchronous and self-timed, on the performance of a RISC microprocessor architecture. It will be shown that a micronet-based datapath can enhance the performance of a microprocessor architecture.

4.2 Comparing Synchronous and Asynchronous Processor Control

The basis for comparison of the two design styles is a simple two-stage pipelined RISC architecture with a simplified instruction set. The justification for the simplicity of the pipeline is the following: isolating the effect of the control paradigm on the datapath is best realised by keeping the latter simple (although the exploitation of pipelining in a micronet processor is discussed in the following chapter); in fact further pipelining interferes with the comparison of datapath latencies in the two designs.

The RISC philosophy of simple control, regular and predictable behaviour and efficient silicon utilisation has been considered ideal for a synchronous control paradigm. The current trend of commercial processors with high frequency clocks are very much in this mould. However, it is difficult to define or find an ideal synchronous processor design since the design itself is inextricably linked with actual implementation delays. An asynchronous control paradigm would be equally applicable to CISC or RISC, however a RISC-style architecture with a simplified instruction set was chosen because of a shorter design time, simpler data paths, and with the corresponding decode/control being hardwired, avoiding any extra level of macro-to-microinstruction translation. This makes it easier to investigate the interactions between the control paradigm and the architecture. (It should be noted that an asynchronously-controlled architecture loses some RISC features e.g. fixed instruction execution times).

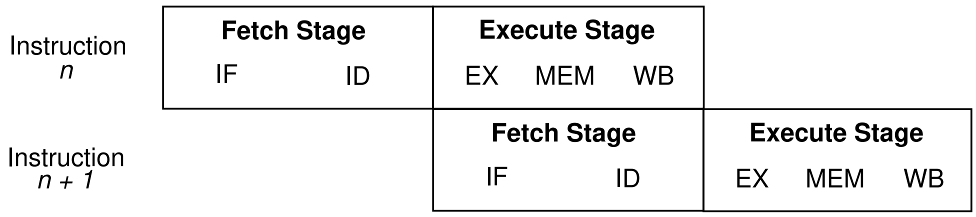


Figure 4–1: The processor pipeline

The two stages, fetch and execute as shown in Figure 4–1, carry out the usual processor operations. The *fetch* cycle involves fetching an instruction or an offset value, and incrementing the program counter; the *execute* cycle involves sequencing data movement and controlling the functional units within the datapath. Thus, the architecture retains the basic RISC features and is a good starting point from which to develop and investigate the suitability of the self-timed paradigm to more complex pipelined processors.

4.2.1 The Two Processor Models

The two processor designs, as illustrated in Figure 4–2, almost share the same functional units and only differ in the design style used to implement their control sequencing. In the *synchronous* microprocessor, the control sequencing is centralised in the *Control Unit* (CU). This unit generates signals for each of the datapath resources (i.e. Fetch Unit, ALU, the Registers, PC Unit and Memory Unit), to control the complete execution of an instruction. In contrast, the control sequencing is decentralised in the *asynchronous* microprocessor. The CU initiates a sequence of actions, and in most cases will no longer take any further part. The respective functional units and their interfaces communicate with each other to complete the task. This reduced complexity of the CU is achieved through the distribution of control by the micronet and the asynchronous mechanisms outlined in Chapter 2. This work naturally extends the theme of early RISC architectures where performance improvements are gained by reducing the

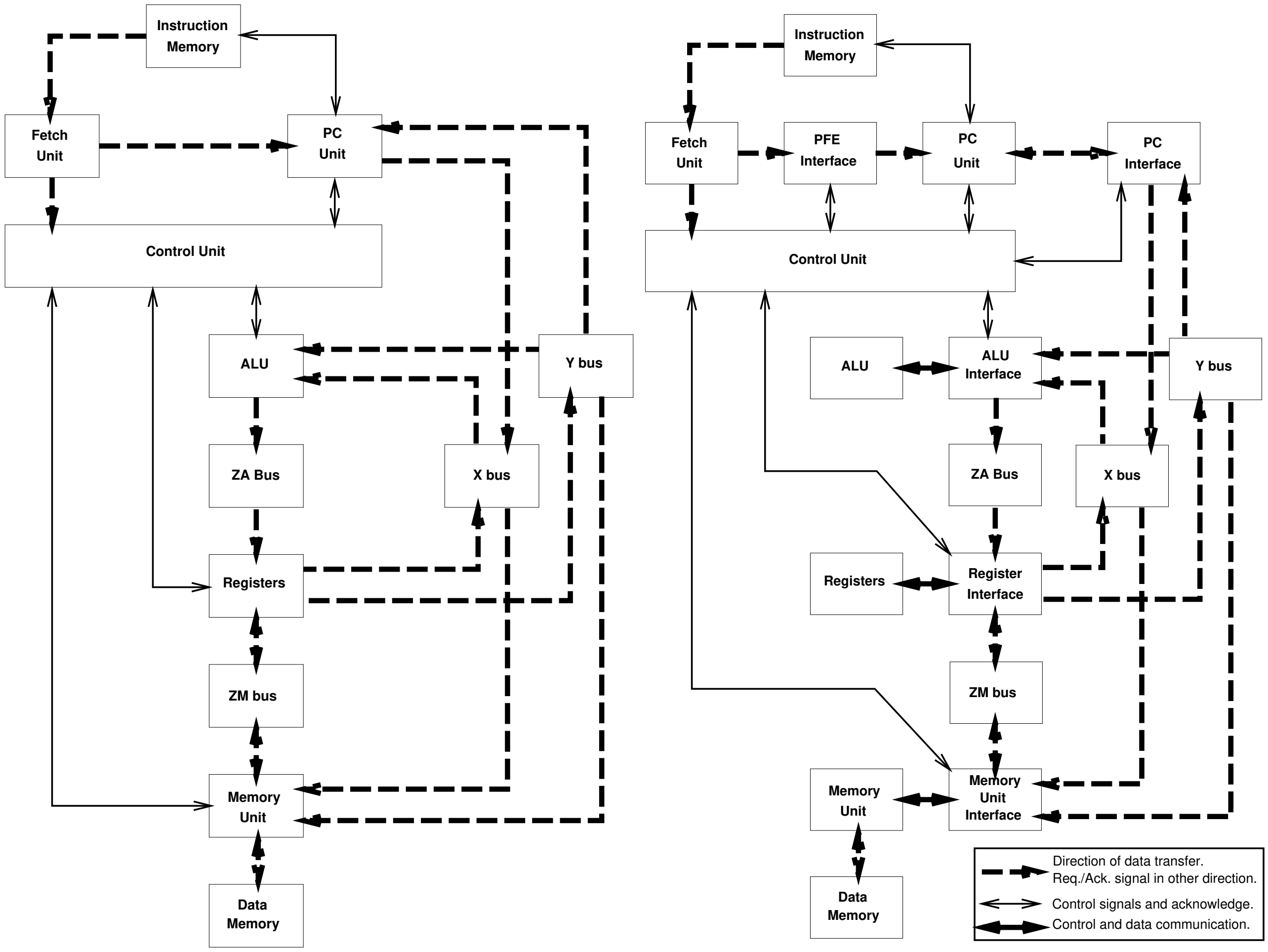


Figure 4–2: The synchronous and self-timed processor models

complexity of the pipeline and simplifying the control. Here the control is simplified even further due to decentralisation.

4.2.2 The Instruction Set

The two designs also share a common instruction set (shown in Table 4–1), which is based on the design in [110]. In the synchronous design, the execution time of each type of instruction is fixed, whereas under asynchrony the execution time of a particular instruction may vary. The different instructions can be

Group	Instruction	Explanation
1	ALU	Rz := Rx ALUop Ry
1	LD	Rz := Mem[Rx+Ry]
1	ST	Mem[Rx+Ry] := Rz
2	LDX	Rz := Mem[Ry+Offset]
2	STX	Mem[Ry+Offset] := Rz
2	LDA	Rz := Ry + Offset
3	STPC	Rz := PC
3	JMP	PC := Ry
4	BRCH	If Cond then PC := PC + Offset

Table 4–1: The instruction set

divided into four categories which highlights the irregular nature of even a simple processor pipeline:

Group 1 – These instructions do not affect the Program Counter (PC) and are therefore independent of the *fetch* stage. The ALU and store (ST) instructions represent the classic single-cycle RISC instructions, with load (LD) instructions taking slightly longer.

Group 2 – These instruction use an offset value which requires an additional fetch from the instruction memory. The current instruction cannot begin execution until the offset has been fetched and placed in the offset register.

Group 3 – These instructions require or modify the current PC, and the next fetch cycle is stalled until the current execute stage is completed.

Group 4 – The branch instruction is a combination of groups 2 and 3. A PC offset is required and the next fetch cycle cannot begin until the current execution cycle completes, i.e. until the branch condition has been resolved and the PC contains the correct value.

4.2.3 The Architectural Components

Figure 4–2 shows the architectural components implemented in both models. The common components are:

1. The Instruction and Data Memory/Cache (IM and DM) which store the program instructions and data, respectively.

2. The Fetch Unit (FU) which fetches instructions from the IM and transfers offset values to the offset register in the PC Unit.

3. The PC Unit (PCU) which contains an adder to increment the PC, the PC register and an offset register.

4. The Control Unit (CU) which initiates the necessary micro-operations in the respective microagents for the given instruction being issued.

5. The Memory Unit (MU) which services the load and store instructions, generates addresses and accesses the DM. This unit has an adder for address calculations. (The input operands must be latched in the unit prior to the unit's operation).

6. The ALU executes arithmetic and logical instructions. It does not have registers on its inputs (or outputs) and operates continuously with the values on its inputs. This allows worst-case operation to complete within the required time.

7. The Register bank consists of 32 registers, three operand read ports to the functional units, and one write port for each of the functional units.

8. X and Y are operand fetch buses, ZA and ZM are write-back buses. The ZM bus is also used as a third operand fetch bus for store instructions.

4.3 The Synchronous Processor

The synchronous model assumes that the control signals are generated exclusively by the control unit, (i.e. the delaying of individual control signal outside the CU to meet any timing constraint is not permitted), using an input clock signal as a timing reference. In synchronous design, the clock period is generally determined by the largest delay in the pipeline. In this example however, the *execute* stage delay varies from instruction to instruction, while the delay of the fetch stage is generally independent of the instruction. Since the latter is always on the critical path, the clock speed was chosen to exactly match the worst-case delay of the fetch stage. However, instead of just viewing each stage as a single cycle, the clock cycle is divided into a number of clock *phases* (four in this example) which mimics a higher frequency clock and reduces idle time by achieving a better approximation to delays. (This allows the modelling of multi-phase clocking as used in modern synchronous designs to improve the temporal granularity).

For the purpose of this study, the synchronisation overheads (as discussed in Chapter 2) are ignored. In practice, they are difficult to estimate as they are ultimately influenced by the clock frequency, technology, fabrication process, routing, chip size and environmental variation.

4.3.1 Synchronous Control

On the first clock edge, the CU initiates a fetch instruction request. The FU then fetches the next instruction from the IM at the location pointed to by the program counter (PC) which is kept in the PCU, and at the same time, the current PC value is incremented. The FU forwards the instruction to the CU just in time for the next clock edge. Now, the CU has the instruction and decoding begins

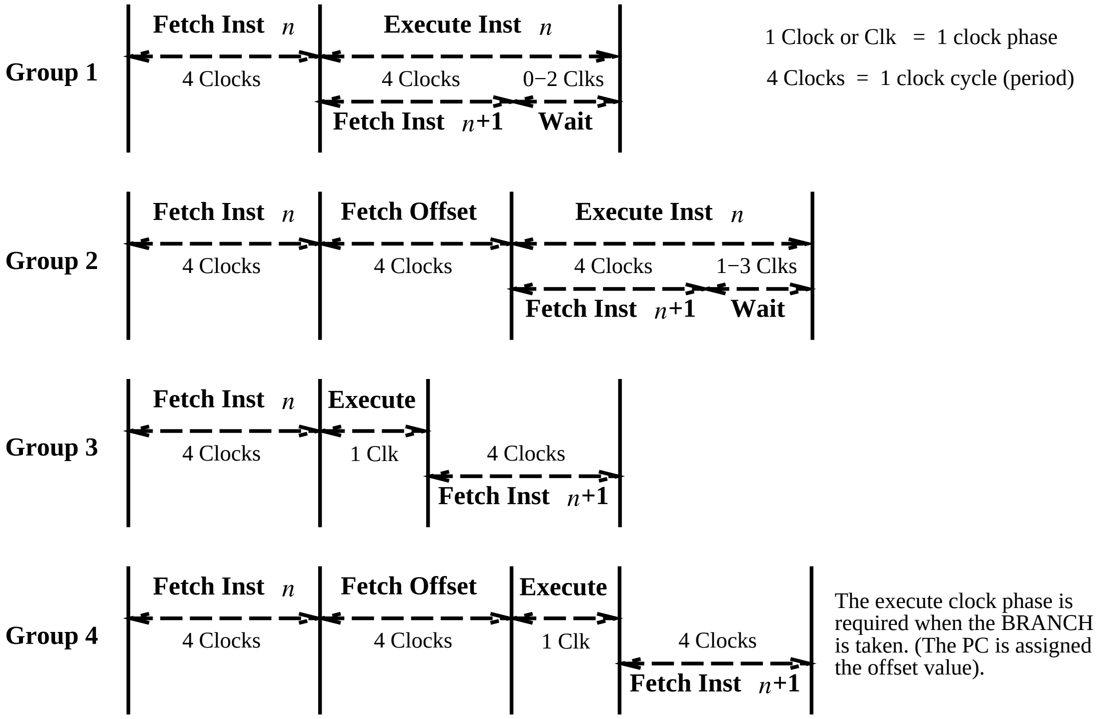


Figure 4–3: Synchronous instruction cycles

while the PC is assigned its incremented value. The CU behaves according to the type of instruction, as shown in Figure 4–3. If an offset is required, then the execution of the instruction is stalled until the offset has been loaded into the offset register. If the instruction is a branch instruction, then it is evaluated while the offset is being fetched. If the branch evaluates to TRUE, then an extra clock phase is required to assign the new PC value. The execute stage latencies vary, taking anywhere between four and seven phases in this example, (the total instruction latencies vary between 8 and 15 clock phases (2 and 4 clock periods)).

4.4 Asynchronous Control and MAP

A micronet-based asynchronous processor (MAP) architecture does not have a global clock signal nor centralised control for the transfer of data between architectural components. Although the processing components (the main functional units) are considered to be identical in the two designs, additional components (the communicating microagents (CMs)) effectively allow the functional units (the functional microagents (FMs)) to locally control data transfer between themselves and their neighbours. In order to exploit data dependent or variable delays, it is assumed that the functional units in the self-timed design can be modified so that they generate completion signals [38] [169].

4.4.1 The Distribution of Control

There are a number of additional components required in the micronet design, as shown in Figure 4–2:

- The PFE interface models a combined interface between the CU, Fetch Unit and PC Unit which aids local control of the fetch pipe. Local control signals (previously routed via the control unit in the synchronous design) between the FU and PCU, coordinate fetching of an instruction while concurrently incrementing the PC or transferring an offset to the offset register (held in PC unit).

- Register, ALU, MU and PCU interfaces are found between their functional units and the buses. These bus interfaces contain the CMs which are responsible for receiving their FM's micro-operation control signals from the CU, returning the corresponding acknowledgement signal, obtaining the operand data for that operation and presenting these to the FU, and if necessary, returning the result of a micro-operation to the correct destination.

A number of protocols have been proposed for both control and data transfers [111] [150] [174] between microagents. In the absence of a clock, the data transmissions have to be encoded to enable the receiver to recognise valid information. Bundled data transfers have been adopted to minimise coding costs [158]. A four-phase handshaking protocol was adopted for both control and bundled data transfer. This allows for a simpler design through the use of various types of Muller C-elements [117] and conventional logic gates. In the case of control signals, although four-phase protocol would be considered twice as expensive compared to a two-phase one, the same efficiency is obtained as two back-to-back, two-phase handshakes by representing two events in each cycle. This is also an efficient option for data transfers since they take place over shared buses, and in any case the second half of the four-phase handshake occurs concurrently with computations. (These issues will be discussed further in Chapter 5). Another advantage of using the four-phase protocol is that it allows components to synchronise phases of an operation, e.g. calculating a next Program Counter (PC) value while using the current PC register value to address memory.

4.4.2 The Rôle of the Control Unit

The CU is still required to sequence tasks for correct datapath operation. Since this control sequencing is decentralised in the micronet, the CU just needs to initiate the sequence of actions, and leaves the respective FUs to communicate to complete the task.

The CU initially requests the next instruction from the Fetch Unit (FU). The FU will then fetch an instruction from the IM based on the current value of the PC, while at the same time signalling the PCU to calculate (increment) the next

PC value. When the FU receives the instruction from the IM, it signals the PCU to assign the calculated value to the PC register, while at the same time checking to see if an offset is required for this instruction. If so, the FU will fetch the offset first, then send the instruction to the CU and pass the new offset value to the PCU. When the CU receives an instruction from the fetch unit it can initiate the next instruction fetch if the current instruction does not use or modify the PC.

Instruction decoding identifies which components or FMs are required to execute the current instruction. The CU communicates with them via the chosen four phase asynchronous communication protocol. Each acknowledgement control signal signifies two events – the first acknowledges the micro-operation request and the second signals the completion of that micro-operation. Should any of the required resources have not completed their previous micro-operation, then the CU must wait until it receives the 'finished' signal, i.e. the previous handshakes have completed. Then the CU can initiate the instruction's execution by informing the relevant microagents (by beginning a handshake on each of the appropriate microagent control signals). Once the CU has received all of the acknowledgements, then the instruction is considered to have been issued. The CU resets the control signals (completes its phase of the handshake protocol) and the instruction issue cycle can begin again. The execution of an instruction is complete when the corresponding control signals have completed their handshakes. Although the current instruction execution is overlapped with the fetching of the next instruction, if the PC unit is involved in the instruction execution it may cause the current instruction fetch to stall.

The registers involved in the instruction execution are informed by the CU as to which buses they have been assigned (derived from the instruction), with respective microagents using the local communication protocol to request their operands. For example, the ALU will assert request signals on both the X and Y buses. This signal (being on a bus) will go to both the register bank and the PC unit. However, only one of them will respond on each of the buses, since the CU will have already notified which components were to be enabled during the current instruction execution cycle.

The Control Signals

The control signals used by the CU effectively consist of a pair of wires: one is the request to, (from now on referred to as the control signal) and the other is the acknowledgement from (referred to as the acknowledgement signal), the FU interface. By using a four-phase handshake protocol the CU can use each of the acknowledgement signals as a status flag (e.g. high to mean busy and low to mean free) for their respective resources. The precise meanings of the control signals and their acknowledgement signals are described below.

As well as the request signal, the control signals to the register bank also consist of the address of the register to which the signal applies. The control signals to the register bank are:

Rx – Identifies the register which should output its contents to the X bus port of the register bank. The corresponding acknowledgement signal is asserted once the register has been accessed (if a register is blocked then it cannot be accessed), and cleared when both the control signal has been de-asserted (following the handshake convention) and the register interface has received the data (i.e. when the interface is ready to transfer the data over the X bus).

Ry – Identifies the register which should output its contents to the Y bus port. The acknowledgement signal is set and cleared as for **Rx**.

Rz – Identifies the register which should output its contents to the ZM bus port. The acknowledgement signal is set and cleared as for **Rx**.

ZMs – Locks the destination register preventing any read access to it. The acknowledgement signal is set when the register is locked and cleared when the register has been written to with data from the ZM bus (i.e. data which has been received from the MU). Note that neither the control signals **Rz** nor **ZMs** can be asserted simultaneously since this could lead to bus contention.

ZAs – Locks the destination register and prevents read access to it. The acknowledgement signal is asserted when the register is locked and cleared when the result from the ALU has been written back to its destination register via the ZA bus.

Other control signals to registers in the PC Unit include:

Rpcx – Outputs the value of the PC on to the X bus. Note that **Rx** and **Rpcx** cannot both be active simultaneously since this could lead to bus contention on the X bus.

Rpcy – The (next) data value on the Y bus is to replace the current PC value.

Rof – Outputs the value stored in the offset register on to the X bus. The PFE interface makes sure that this register holds the correct value. As before, the **Rx** and **Rof** cannot both be active simultaneously.

The control signals to the functional units (microagents) can take one of two forms. Firstly, the control signals can contain the instruction opcode (or some part of it) which is decoded locally by the functional unit itself (as in the case of the ALU's control signal **AU**). Here the local decoding is overlapped with the instruction's operand fetch. Secondly, if the decoding costs are small and do not increase the CU delay, it may be possible to decode the opcode and use dedicated control signals for particular (micro)operations within a functional unit. Control signals to the MU are **MU1** for a load (LD) instruction, **MU2** for the store and **MU3** for the address calculation instruction. In this case the cost is generally hidden by the instruction issue handshake of the previous instruction.

4.4.3 Data Transfer

Data transfer is request-driven, e.g. a functional unit which requires an operand will assert a request to the register. The register will in turn send the data on the bus, the reception of which is acknowledged by the functional unit's interface by de-asserting the original request. Thus allowing the register to release the bus. Generally, this ensures that resources (registers and buses) are utilised for no longer than is necessary. The register control signals together with the handshaking protocol prevent bus contention occurring.

4.5 The Performance Results

All the functional units in Figure 4–2 were based on a 1.2 μm CMOS implementation process. Their timing characteristics were extracted from a post-layout simulation tool within a commercial VLSI design package called SOLO 1400 [50] and used in the PEPSÉ simulation models of the processors. Neither layouts nor transistor size optimisations for improved performance [26] were considered.

The performance of the instruction set outlined in Table 4–1 is summarised in Table 4–2. In the simulations, every effort was made to make the comparisons between the two design styles as fair as possible. While the chosen implementation process is not state-of-the-art, no commercial design tools nor sufficient commercial processor layout information was available upon which to base an accurate comparison. Also, commercially available synchronous architecture generally contain a number of engineering and design "tricks" specific to particular implementations of a design.

Group	Instruction	*Synchronous Design*		*Asynchronous Design*		Speed Up
		Inst.Exec. Time (nS)	Clock Phases	Inst.Exec. Time (nS)	Datapath Exec.Time	
1	ADD	36	4	26	17	1.38
1	LD	54	6	34/26	34	1.58/2.07
1	ST	36	4	26	14	1.38
2	LDX	99	11	60/55	60	1.65/1.8
2	STX	81	9	55	40	1.47
2	LDA	81	9	55	43	1.47
3	STPC	45	5	32	20	1.40
3	JMP	45	5	32	9	1.40
4	BRCH F	72	8	59	32	1.22
4	BRCH T	81	9	63	42	1.28

Table 4–2: Synchronous versus asynchronous performances

The results of the comparison of instruction execution times under the two control philosophies are shown in Table 4–2. The *Instruction Execution Time* (IET) represents the time between issuing the current instruction and the next, i.e. the effective cost for fetching and evaluating each instruction, taking into account the two staged pipelined nature of the processors. In the synchronous case, the minimum IET is 36nS (the clock period) which is equivalent to the delay of the fetch stage. The fetch stage delay is 26nS in the micronet design, which considers both the average timings and the self-timed overheads.

The *Datapath Execution Time* (DET) is the average duration between the CU initiating an instruction and its completion, i.e. the instruction latency within the (execute stage of the) micronet datapath. The IET is the maximum of the fetch stage delay and the execute stage delay (DET). The DET is of particular interest when it is larger than the fetch stage delay (26nS) since this means that the CU might be able to exploit some concurrency by being able to overlap the execution of more than one instruction within the datapath. If the following instruction is independent, then the effective IET of the previous instruction will be the smaller value ($IET_{unrelated}$). Otherwise, in the presence of structural or data dependencies, the larger value applies ($IET_{related}$). When comparing execution times between the two design styles for the load (LD) and load with offset (LDX) instructions, the $IET_{related}$ value should be used, because in the synchronous case wait states have been inserted in these instructions as the CU must assume the worst-case situation. Although, in general, this suggests that MAP can exploit some data-dependent concurrency, the synchronous processor's CU could test successive instructions for structural and data dependencies at the expense of increasing the complexity and delay of the unit. The asynchronous design can take advantage of any independence between instructions without testing, since the handshaking mechanism will prevent erroneous behaviour should such a dependency exist.

In the self-timed design, the IETs of the instructions are limited by the fetch stage delay. In fact the speed-up in these cases virtually represents the ratio of latency between the two instruction fetch pipes. Even though the synchronous fetch pipe has a perfectly matched clock it is still limited by worst-case delays and an inability to generate control signals at precise times due to its centralised control.

These speed-ups show that it is indeed possible to achieve performance improvements under an asynchronous control paradigm. Since all of the instructions show improvement, a program consisting of these instructions will therefore be expected to execute faster. Furthermore, it is in the nature of the self-timed CU to initiate instructions as soon as possible. This can only be achieved at run-time. However, the timing characteristics used for the synchronous CU are fixed at design time.

The preliminary conclusion from these results is that one can observe an improvement in performance of the asynchronous control mechanism over their synchronous equivalent, when the individual instruction execution times are compared. The MAP architecture uses circuits that generate completion signals [169] and therefore benefits from exploiting actual component delays. The magnitude of any improvement is limited by a number of factors. The two important ones are: the architectural design, where some sort of decoupling is required between the two stages since each of the stages can stall waiting for the other; and the difference between typical and worst-case delays which is influenced by component design.

4.6 Discussion

MAP implementations are robust to variation in physical parameters and can adjust to variations due to data-dependent operations. For instance, the time to add two integers using a ripple-carry adder varies with the length of the carry chain. The clock period of a synchronous implementation has to be adjusted for the worst case, and therefore a synchronous version takes time proportional to the number of bits of the operands. On the other hand, an asynchronous ripple-carry adder computes in time which is on the average proportional to the logarithm of the number of bits [60] [109]. This is at a cost of detecting the completion of the operation locally. However, the overheads of the handshake mechanism can be hidden in micronets, as will be shown in the following chapter.

If the duration of all of the operations were constant and known precisely, then the sequencing could be implemented efficiently with a global clock and centralised control, since this is sufficient to signify the end of a computation and start of the next one. Timing relies on the physical and environmental parameters of the design. Designers, being aware that their knowledge of both the physical properties of the devices and the runtime behaviour of the circuits is imperfect, have to lengthen the clock period to account for an error margin in the evaluation of the duration of a computation step. This error margin is becoming a significant proportion of the operating clock period and actually leads to inefficiency. Furthermore, delays have to be matched by a discrete number of clock cycles which gives rise to idle times which can become quite significant. Incorporating a variable period clock [39] or using a faster clock leads to diminishing returns; increases the design complexity without necessarily improving performance significantly. In fact, increasing clock frequency has been the popular solution although such signals induce noise, and their distribution is difficult and subject to skewing, as discussed in Chapter 2.

For complex computations with data dependencies, asynchronous design has the advantage of exploiting the best-case delay, whereas synchronous solutions have to adjust to the worst-case. Furthermore, data flowing in a network of stages rather than a linear pipeline may not encounter the component with the largest delay (slowest stage), e.g. not all instructions need to use a shifter, and therefore will not even be hindered by the slowest operation (which itself may not be executing at the time).

4.7 Summary

This chapter has described two similar microprocessor designs which differ only in the control strategy. The architecture incorporates the basic features of RISC without complicating issues such as pipeline hazards and provides a good foundation from which to develop and investigate the suitability of the self-timed paradigm for more complex pipelined processors. The synchronous design incorporates conventional centralised control mechanisms. The sequencing of instructions is controlled centrally in the *control unit* which generates the control signals for each of the other components in the datapath with timing provided by a clock signal. The clock period is fixed by the largest possible delay within a stage in the pipeline. In an asynchronously controlled microprocessor, control sequencing is decentralised amongst the datapath's functional units. The execute unit just initiates a sequence of actions, and in most cases will take no further part. The corresponding components will then communicate between themselves via request and acknowledge handshakes in order to complete the task. This allows an operation to proceed at a rate determined by local, variable delays and not by a delay which is fixed pessimistically.

This alternative control paradigm is realised through a micronet and the main concern in this chapter has been with the exploitation of actual datapath delays in micronet-based processors. Results obtained via simulation have been presented for the performance of an instruction set for two design styles of microprocessor. These indicate an improvement in performance (on average) for the self-timed design over the synchronous equivalent. These results only represent the performance gain per instruction. Since all the instructions have shown improved execution times, the execution time of a program containing an average instruction mix will also be better. The magnitude of these results really depends on the type of operation being carried out and the design style of the functional units (e.g. ALU design). The speed up reported here does agree with other related work by Dean [39] and predictions by Ginosar [63].

Further improvements in performance are possible by taking advantage of instruction-level parallelism (as in most commercially available RISC processors). The MAP's control unit can exploit some execution concurrency if it can issue the following instruction before the previous one has finished. This incurs no extra cost in this design unlike a synchronous processor's control unit. Allowing concurrent instruction execution introduces pipeline hazards [72] into the design. The following chapter examines the modifications to the design of the MAP architecture which exploit more fully the underlying self-timed control paradigm, for exploiting ILP.

Chapter 5

The Control Paradigm and the Architecture

5.1 Introduction

The previous chapter compared a synchronous RISC processor architecture with its asynchronous equivalent. Centralised control and synchronous data communication were replaced by distributed control and asynchronous communication without the higher levels within the computer system perceiving any changes. It was shown that an asynchronous *control paradigm* could indeed improve the performance of the instruction set for a given processor architecture. That design experiment only attempted to improve the execution times of individual instructions, made possible by the micronet's ability to exploit actual component delays as well as hiding some of the handshaking overheads. However, in order to realise the full potential of this asynchronous design style, this chapter attempts to highlight the ease with which a MAP architecture can be modified to exploit Instruction-Level Parallelism (ILP). Refinements are made to a modified version of the micronet processor architecture described earlier, to efficiently improve performance through the increased utilisation of the datapath resources and to exploit ILP without significantly increasing control costs. In fact, ILP is used to effectively hide the remaining overheads due to asynchronous control.

5.2 Exploiting Instruction-level Parallelism

Speeding up the execution latencies of instructions is one approach to improving performance. An alternative is to execute more than one instruction at the same time. Exploiting ILP [84] can be achieved either by issuing several independent instructions per cycle as in superscalar or VLIW architectures, or by issuing an instruction every cycle, where the cycle time is now shorter than the times for any of the operations, as in (super)pipelined architectures. Furthermore, these two approaches may also be combined.

The superscalar principle relies primarily on exploiting spatial parallelism, which is achieved by running multiple operations concurrently on duplicated hardware. In contrast, pipelining relies on exploiting temporal parallelism by overlapping multiple operations on common hardware and operating with a faster clock. Note that ILP is limited by data dependencies between instructions, structural dependencies and also control transfers in pipelined architectures.

Most, if not all, processor architectures are pipelined (to some degree) since it is considered the most cost effective of the two alternatives. However, the limits on this type of concurrency have meant that modern processor designs need to consider the more expensive form as well [40] [42]. This chapter concentrates on implementing asynchronous “pipelines” for exploiting ILP (both temporal and spatial) as a number of control issues resulting from data and structural dependencies between instructions have to be addressed efficiently. Since a good instruction schedule (generated statically) to avoid such dependencies is not always possible, techniques are required to resolve them at run-time. Within synchronous datapaths, structural hazards are normally avoided in hardware by using a scoreboarding mechanism and data dependencies are resolved by using either hardware or software interlocks [70], which adds to the control complexity and cost. Data Forwarding is a technique commonly used in pipelined architectures to minimise the cost of functional unit (FU) stalls due to data dependencies, by redirecting data being written to registers to the waiting functional unit [163]. In synchronous ILP designs, the cost of maintaining correct operation increases the complexity of control which in turn adversely affects the clock period and therefore the performance. However, an asynchronous datapath which is designed using micronets can use the existing handshaking mechanisms, together with the simple locking of registers, to achieve the same effect with trivial hardware overheads. Exploiting concurrency in a micronet architecture is aided by the distributed nature of the control strategy and by the fact that data movement is controlled locally. Previously, it had been considered expensive to pipeline decoding, but here this is no longer the case since control and decoding are distributed amongst architectural components. As a consequence, implementing asynchronous superscalar or superpipelined architectures is relatively straight-forward, and this will be discussed briefly in Chapter 7.

In practice, all instructions do not necessarily have identical execution times and thus the results of instructions may be ready out of program order. Enforcing in-order write-back to registers is inefficient for performance, since this can effectively stall functional units and thereby increase the evaluation time of instructions. Out-of-order instruction completion can be supported in synchronous designs, but at a non-trivial cost [40]. In contrast asynchronous designs, as proposed in this work, can relax the strict ordering of instruction completions and thereby further exploit ILP. The effect is to increase the utilisation of the functional units by reducing their stalls. By exploiting both ILP and actual run-times of instructions, better program performances can be achieved on asynchronous processors, and this will be demonstrated in greater detail in the rest of this chapter.

5.3 Design Goals

A goal of early synchronous RISC architectures was to achieve an execution rate of one instruction per machine cycle. In simple architectures, like the design in Chapter 4 which followed the sequential mode of program execution and avoided hazards, this meant an instruction would complete its execution before the next one started. Such processors did not have a pipelined execute stage and either the choice of instructions within the instruction set had to be restricted by the requirement that the execution time of each instruction be equal to a single (and in later RISC architectures – a fixed multiple of the) clock period (in order to achieve a certain performance or MIPS rate) or that the clock period was determined by the execution time of the slowest instruction. Remember that the clock period itself is determined on the basis of conservative estimates of component delays. Therefore all instructions are viewed as executing in the same time irrespective of their actual delays even though most instructions will actually complete in some fraction of the clock period. Also, in practice, different instructions generally require different resources and even the same instructions can have different execution times. All of this leads to poor utilisation of expensive resources. Although pipelining has gone some way to redressing this, the technique itself introduces inefficiencies: stage balancing problems, for example the von Neumann bottleneck makes it difficult to match the cost of fetching an instruction with its execution. Whereas the RISC philosophy was concerned with the efficient usage of silicon real estate, the goal of the micronet control paradigm is more efficient utilisation of the functional units over time.

5.4 An Asynchronous ILP Processor

The structure of a processor architecture is determined by the number and type of components or functional units and their connections. Pipelining is a control technique for exploiting temporal ILP. The first MAP architecture under investigation is a modified version of the one described in Chapter 4. The functional units are identical to those used in the previous design, with the exception of those in the fetch stage. The modifications in the execute stage focus on optimising the control and data handshake protocols to improve the control sequencing and supporting ILP. These modifications have been implemented in a series of refinements and at each refinement, their effect on program performance is measured. An adequate set of instructions has been implemented in each refinement step to highlight the effects of the modification.

The results in Chapter 4 have clearly shown how the asynchronous processor's performance is affected by the fetch stage. It is therefore necessary to reduce the fetch stage delay to less than the smallest execution cost in order to ensure that the execution pipe is kept busy. (Note that the fetch cost, being independent of the instruction set, is more a function of the memory technology which allows the overall processor performance to be traded off with the financial cost of the instruction memory/cache). Also, the amount of concurrency that can be exploited in such an architecture is severely restricted by the fact that the PC has to be available to both the fetch and execute stages. The work in this chapter focuses on the the datapath within the execute stage. In order to improve resource utilisation and expose maximum concurrency, a number of minor architectural modifications are made to the design described in the previous chapter, to create the base architecture upon which further (control) improvements will be made.

5.5 A Micronet Architecture

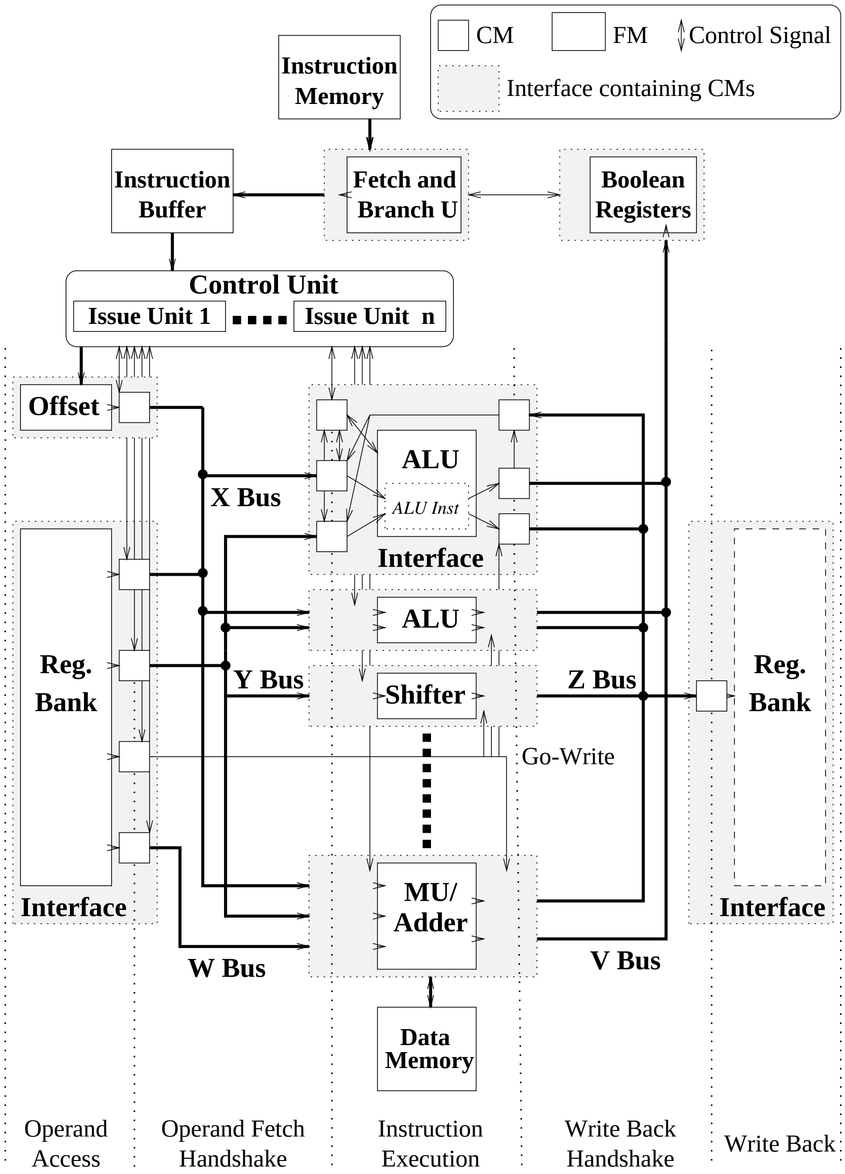


Figure 5–1: A typical micronet-based processor architecture model

Figure 5–1 illustrates the functional units which might constitute a typical MAP architecture. The intention is not to focus on the functional units themselves, but rather on their asynchronous control using micronets and the resulting performance improvements. The number of units and their functionality can be changed without any side-effects. The base architecture under study is comprised of the following units:

1. As previously, the Instruction and Data Memory (IM and DM) or Cache store the program instructions and data, respectively.

2. The Fetch and Branch Unit (FBU) fetches instructions from the IM, executes control transfer ones and places the others in the instruction buffer.

3. The instruction buffer is an asynchronous queue which effectively decouples the fetch stage from the execute stage.

4. The Control Unit (CU) initiates the necessary micro-operations in the respective microagents for a given instruction.

5. The Memory Unit (MU) services the load and store instructions, generates addresses (using its own adder) and accesses the DM.

6. The ALU executes arithmetic and logical instructions.

7. The Register bank consists of number of registers (32), three operand read ports to the functional units, and a write port for the ALU and the MU.

8. The Boolean Register Bank holds flags which are used to resolve branch conditions.

9. X and Y are operand fetch buses and V is the boolean flag write-back bus. The Z bus is initially used as both an operand fetch bus (labelled W in Figure 5–1) and a register write-back bus.

5.5.1 Modifications to the Fetch Stage

The combination of an unbalanced two stage pipeline and the implementation of certain instructions (particularly control transfer ones) could cause the execute stage to often become starved of instructions. This will have a detrimental effect on the exploitation of concurrency and efficient utilisation within the execute stage of the datapath, and therefore this behaviour has to be improved. Firstly, all PC-related instructions are either executed in a new unit called the Fetch and Branch Unit (FBU) or removed from the instruction set altogether. The FBU is responsible for fetching instructions from the instruction memory or cache and processing control transfer instructions. This unit filters out unconditional branches and updates the PC directly. The branch target address is copied to a register after which branch prediction schemes similar to those employed in synchronous designs can be applied. Although the removal of the execution of PC-related instructions from the execute stage may be seen as the influence of the control paradigm on the processor architecture, this feature has already been incorporated in high performance synchronous designs (e.g. [40] [44] [155]). The problem is related to the fact that it is difficult to exploit parallelism when a resource is being used in separate stages within the datapath.

As described in Chapter 2, pipeline stages have a producer-consumer behaviour. If two stages have varying delays such that their worst-case delays alternates between them, then the pipeline latency will be the sum of the two worst-case delays. If the stages are decoupled from each other by an asynchronous queue which stores the predecessor's results, then the stall time of the stage is reduced and throughput improved. An instruction buffer/window has been implemented to hold instructions pending execution. Situated between the two stages, the buffer relaxes the synchrony between the FBU and CU, allowing each stage to proceed at its own rate without hindering the other until the buffer becomes either full or empty. Thus, the decoupling of the fetch stage from the execute stage can reduce the amount of time the control unit is starved of instructions. The FBU continuously fetches instructions and places them in the buffer until either the buffer is full or the unit stalls waiting to resolve a conditional branch (control transfer). Unconditional branches will be executed by the unit, updating the PC immediately. The problem of control transfer resolution is, however, made more difficult. Although this is similar to the problem faced by deeply pipelined synchronous processors, the effect of the buffer is to introduce a variable number of pipeline stages between the instruction being fetched and the instruction being issued (executed). Ignoring control transfers, implies that the current PC value will no longer be just one (or a constant number) ahead of the PC value of the instruction being executed, which makes it difficult to use the PC value in the execute stage. The use of branch prediction schemes to prevent stalling the pipeline and conditional instruction execution as a solution to malpredicted branches can be employed without affecting or being influenced by the control paradigm (see Chapter 7). The instruction buffer has an additional use in more advanced designs which will also be elaborated in the same chapter.

5.6 The Control Refinements

The following sections discuss the refinements made in a number of steps to the execute stage of the base MAP design shown in Figure 5–11[1]. These refinements highlight the ease with which the micronet model can both efficiently exploit ILP and obtain good functional unit utilisation without the difficulties normally encountered in synchronous datapath design (e.g. implementing hazard avoidance, data-forwarding or balanced pipeline stage design). Control is distributed at each refinement step to the functional units, and improvements, if any, in the execution of sample programs are recorded. An architecture, as illustrated in Figure 5–1, is composed of a network of microagents (denoted by solid boxes) which are connected via ports. The Functional Microagents (FMs)

[1] Figures 5–11 to 5–18 can be found at the end of this chapter, from page 133 onwards.

perform micro-operations which are typical of a datapath. On each port of a FM is a Communicating Microagent (CM) which is responsible for communication among the FMs, and with the Control Unit (CU). The FMs are effectively isolated and only communicate through their CMs, and can therefore be modified without affecting the rest of the micronet. The modifications to the datapaths are modelled using micronets as shown in Figures 5–11 to 5–18. These versions aim to exploit the fact that the microagents operate concurrently, each executing one micro-operation at a time. In Figure 5–11, for example, four microagents can operate in parallel in the operand access stage; followed by three pairs in the operand fetch handshake stage; two in the instruction execution stage; two pairs in the write-back handshake stage; and two in the write-back stage.

5.7 Measuring Improvements in Performance

The two parameters which affect the performance of programs in asynchronous pipelines are the *latency* of the microagents, which is defined as the time between initiating the micro-operation and the result being signalled as available; and their *cycle time*, which is the minimum time between successive initiations of the same micro-operation, i.e. throughput. The two parameters have the same value in a synchronous pipeline, with the cycle time being determined by the latency of the slowest stage. The difference between the two values may be viewed as the overhead due to asynchronous protocols and a good design should endeavour to minimise it. This is achieved in micronets by overlapping the phases of the communication protocol in the CMs with operations in the FMs, thus hiding the overhead through concurrent operations. The effectiveness of this method is gauged by measuring the utilisation of FMs when exercised by test programs composed of the appropriate, identical instructions. Metrics are now introduced for characterising the performance of micronet datapaths.

Minimum Micronet Latency (MML) is the time between asserting the control signals (i.e. initiating an instruction issue) and receiving the final acknowledgement of the instruction's completion. From the CU's point of view, this is the shortest execution time (latency) through the micronet (ignoring any stall time due to busy resources) for a particular instruction. This value influences when successive data dependent instructions can begin their execution. Note also, that this metric is not the same as the Datapath Execution Time (DET), as used in the previous chapter, which is just the time taken for the instruction's result to reach its destination (i.e. it does not include the time to signal the instruction's completion).

Instruction Cycle Time (ICT) – In asynchronous pipelines, which usually have non-uniform stage delays, the time between successive instruction issues is influenced by the slowest stage *currently* active in the pipeline. The ICT is the time between two identical instruction issues once that instruction's pipeline is full. This metric is the sustainable rate at which a particular type of instruction can be issued. The upper bound on this value is determined by the cycle time of the slowest microagent on the instruction's path. (Instructions are executed by following the particular paths through the micronet). Note that this is not a strict upper bound since the time between these instruction issues could increase because of contention for a shared resource (caused by the concurrent execution of a different instruction). For example, a different functional unit starts using the write-back bus causing another instruction in the current instruction's micropipeline to stall. In practice, if this only happened occasionally, it may not affect the ICT since the elasticity of the micronet may absorb the effect.

Program Execution Time (PET) is the actual execution time of a program. As this time is reduced, component utilisation will increase (assuming the amount of work stays the same). For a micronet executing a stream of identical instructions, the PET can be approximated to:

$$(n-1) \cdot ICT + MML + overheads \tag{5.1}$$

where *n* is the number of instructions and the *overheads* are the costs associated with the initial instruction fetch startup. Equation 5.1 is obviously related to the synchronous equivalent where the ICT would be equivalent to the clock period and MML to the pipeline latency, i.e. the clock period multiplied by the number of stages in the pipeline. Note that average values have been used for modelling purposes but in practice it is likely that both the ICT and MML of an instruction would vary.

ALU Utilisation – The percentage of the program execution time (excluding the initial instruction fetch time) for which the ALU performs useful computation. Utilisation measurements are important for two reasons: firstly, they are a measure of efficient functional unit usage, greater efficiency leads to improved performance; secondly, high utilisation can also indicate potential bottlenecks within the design. Although adding another resource may improve program performance and reduce the utilisation (an architectural design trade-off), this work advocates that given a set of architectural resources, an asynchronous control paradigm is better able to utilise them.

MU Utilisation – Same as above, but for the Memory Unit (MU).

Register Utilisation – Same as above, but for the Register Bank. This figure is useful since in the nature of RISC architectures all data must be moved via the register bank which could pose a potential bottleneck.

ALU Interface Utilisation – The percentage of the execution time (excluding the startup latency) during which the ALU's CMs are busy.

MU Interface Utilisation – Same as above, but for the Memory Unit Interface.

Register Interface Utilisation – Same as above, but for the Register Interface.

Program Minimum Instruction Issue Cycle Time (MIICT) is the minimum time between successive instruction issues, which gives a measure of the maximum possible issue rate for a given program. The ratio of the largest MML and smallest MIICT is an upper bound on the number of instructions which can potentially execute concurrently in the datapath.

Maximum FM Utilisation – The upper bound on the FM utilisation for a particular instruction is the ratio of the FM micro-operation latency and the ICT for that instruction. Therefore, architecture designs should aim to reduce the ICTs of instructions to that of their FM micro-operation delays. Given that the ICT is determined by the slowest delay on the instruction's path, optimal utilisation can only be achieved when the FM is the slowest microagent. (In terms of program execution it is assumed that only FMs do useful work and the other operations are effectively the overheads associated with the architectural design).

5.7.1 The Test Programs

The feasibility of taking advantage of actual delays rather than assuming the worst-case values depends on the difference between the actual and worst-case delay being larger, on average, than the overheads due to asynchrony. If the asynchronous overheads were to be hidden then asynchrony would always have a performance edge. The successive refinements aim to show that the exploitation of fine-grain ILP can be used to hide these overheads.

The actual performance of the architecture is determined by delays of the components. It is demonstrated that the maximum attainable performance approaches the maximum possible performance of the architecture. The FU latencies are chosen to be constant – the average execution time, to capture the essential behaviour of micronets.

The micronets in Figures 5–11 to 5–18 were exercised by programs with a mixture of LD, STR, and ALU instructions (see Appendix C for more details). The Alu, Load and Store test programs (ATP, LTP, STP) measure the maximum attainable utilisation of their respective FMs. They contain repetitions of either ALU, LD or STR instructions, so that only structural dependencies exist between instructions (in effect setting up a static pipeline or a fixed path through a network of components). The number of instructions in the test programs are sufficient to fill the pipeline, i.e. enough instructions exist to allow the CU to achieve a steady issue rate. The Hennessy Test (HT1) consists of a mix of the three instructions, but without any data dependencies, which exercises the spatial concurrency and out-of-order completion, for a particular schedule devised by the compiler. HT2 is a variant of HT1 but with data dependencies, which exercises the data forwarding mechanism as well. This program represents a "typical" basic block of compiled code (actually a line of code in C from [70]).

In the following sections, the refinements which were made to MAP in order to exploit ILP through the distribution and decentralisation of micronet control have been described together with the performance results that have been measured in the PEPSÉ environment.

5.8 Refinement Step 1 – The Base Case

Figure 5–11 illustrates a naïve implementation of an asynchronous datapath which does not as yet fully exploit the properties of micronets. Refinement Step 1 only exploits the actual execution timings of micro-operations. The execution of each instruction requires a predetermined set of micro-operations, each initiated by signals from the CU. These are four-phased controls whose acknowledgement signals are used as status flags for mimicking a scoreboarding mechanism. The micro-operations for an instruction are initiated as soon as possible by asserting the necessary control signals. The receipt of an acknowledgement confirms that the associated micro-operation has begun and the initiating control signal is de-asserted. The instruction is said to be issued once all the asserted control signals have been acknowledged, and the next instruction issue can begin.

These micronet control signals are described in greater detail below, with the micro-operations required by each instruction outlined in Table 5–1:

Rx – This signal identifies the source register for the X Bus and the corresponding acknowledgement is asserted once the register has been accessed, and cleared once the data has been transferred to the operand fetch handshake phase.

Ry – This is the same as above but for the Y Bus.

Rz – This is the same as above but for the Z Bus when used for fetching operands.

Rof – This is similar to **Rx** except that it is used to access the offset register, the contents of which are output on to the X Bus. **Rof** and **Rx** cannot be asserted simultaneously since they both require the X bus.

AUs – This signal identifies the next operation of the ALU and the corresponding acknowledgement is asserted when the interface is ready to fetch the ALU's operands from the register and is cleared when it initiates the write-back handshake.

MC1 – This signal identifies a load instruction to the MU and is asserted and cleared in the same manner as **AUs**. Other signals exist for both the store

(STR/STX) and the address calculation (LDA) instructions but these have been omitted for the sake of brevity.

ZAs – This signal identifies the destination register for writing back the result of an ALU operation via the ZA bus and the corresponding acknowledgement signal is asserted when the register is ready to receive data and cleared once the data has been written back.

ZMs – This is the same as above, but for data written back from the MU via the ZM bus.

Instruction	Required Micro-operations
ALU	Rx Ry AUs ZAs
LD	Rx Ry MU1 ZMs
ST	Rx Ry Rz MU2
LDX	Rof Ry MU1 ZMs
STX	Rof Ry Rz MU2
LDA	Rof Ry MU3 ZMs

Table 5–1: The micro-operations required for instruction execution

Figures 5–11 to 5–18 illustrate the micronet model through the series of refinements. For each refinement step, they identify the stage during instruction execution when each of the acknowledgement signals is generated. The timing diagrams correspond to the execution of a load followed by an add instruction which highlights the relationship between the control signal transitions.

In Refinement Step 1, all the micro-operations for an instruction are initiated at the same time and the next set can only be initiated after the completion of the micro-operations of the current instruction. This effectively serialises the instruction execution, as illustrated in the timing diagram in Figure 5–11. As an example, the behavioural description of the CU issuing a LDA instruction is given in Figure 5–2. In successive refinements the rôle of the CU will be diminished by distributing the control of the micronet to local interfaces, with micro-operations being initiated individually as early as possible.

CASE instruction
:
LDA : SEQ

wait until ($\overline{\mathrm{RxA}}$. $\overline{\mathrm{RofA}}$. $\overline{\mathrm{RyA}}$. $\overline{\mathrm{RzA}}$. $\overline{\mathrm{AUA}}$. $\overline{\mathrm{MU1A}}$. $\overline{\mathrm{MU2A}}$. $\overline{\mathrm{MU3A}}$. $\overline{\mathrm{ZAsA}}$. $\overline{\mathrm{ZMsA}}$);
Wait until the handshake cycle of all control signals have been completed, by testing the input acknowledgement signals.
STATE 1.
assert (Rof, Ry, MU3, ZMs); Initiate instruction execution.

wait until (RofA . RyA . MU3A . ZMsA); Wait until the Control Signals have been acknowledged.
STATE 2.
deassert (Rof, Ry, MU3, ZMs); Instruction issued.

Incoming Offset Register Acknowledgement Signal	- RofA (Active phase), $\overline{\mathrm{RofA}}$ (reset phase).
Outgoing Register Offset Control Signal	- Rof

Figure 5–2: Issuing an LDA instruction in Refinement Step 1

Performance Results

Instruction	ICT	MML	Max. FM Utilisation
ALU	24nS	24nS	16.67%
LD	43nS	43nS	53.49%
ST	23nS	21nS	42.85%

Table 5–2: Instruction execution for Refinement Step 1

The ICT value for an instruction is determined by its slowest microagent control signal handshake, since the instruction issue is serialised. The results in Table 5–2 show that the Instruction Cycle Time (ICT) is equal to the Minimum

Test Programs	Alu Test	Load Test	Store Test	HT1 & HT2
Program Execution Time	175nS	308nS	164nS	143nS
MIICT	24nS	43nS	21nS	21nS
ALU Utilisation	16.57%	0%	0%	8.39%
MU Utilisation	0%	53.31%	39.87%	22.38%
ALU Interface Utilisation	78.7%	0%	0%	39.86%
MU Interface Utilisation	0%	88.08%	82.91%	38.46%

Table 5–3: Execution of the test programs on Refinement Step 1

Micronet Latency (MML)[2] (except for the ST instruction), which is not surprising as instructions execute sequentially but only take as long as is necessary. The higher value for the ST instruction is due to a handshake delay, which in the case of the LD instruction is hidden by the write-back stage (discussed later in this section). Although there is no explicit pipelining of the datapath, different phases of the handshaking may occur at the same time, e.g. a CM may initiate a handshake with another CM while completing one with its FM. This is reflected in the interface utilisations shown in Table 5–3.

Also shown in Table 5–2 are the figures for the maximum FM utilisation which represents the proportion of the MML taken by the FM to complete its operation. As predicted, the execution times of the test programs in Table 5–3 are the sum of their individual instruction execution times together with startup overheads. It is observed that the utilisations achieved for the FMs (in Table 5–

[2]The values given here differ from those in the previous chapter due to the following reasons: DET and MML are slightly different measures (see pages 74 and 89); changes to the CU caused by the architectural modifications described earlier in this chapter; and a different choice of design process and cell library has been used to implement the datapath components (see page 58).

3) are very close to their upper bounds (in Table 5–2) which demonstrates that asynchronous control using a micronet can be efficient.

The Store Instruction's Cycle Time

The MU only receives the next control signal, i.e. its next operation once it has completed the current instruction. Only then can the MU make a request to its interface for the necessary operands. The increase in cycle time is due to the operands waiting at the interface for this request because of the shared use of the Z port (as both an input and output). This delay is effectively hidden by the write-back operation in a load instruction.

5.9 Refinement Step 2 – Exploiting Multiple Write-back Buses

An instruction's micro-operations are still asserted and de-asserted collectively, but as soon as all the *relevant* signals become ready, i.e. without having to wait for earlier unrelated micro-operation handshakes to finish. This introduces overlap between successive instructions which require different micro-operations. This feature of the micronet helps to exploit even finer-grained spatial concurrency between instructions than previously possible. In Figure 5–12, while instructions share the operand fetch resources, the two FMs and their write-backs (WBs) can operate concurrently. This implies that there is scope for out-of-order completion of instructions, which introduces pipeline hazards, such as Read-after-Write (RAW), Write-after-Write (WAW) and Write-after-Read (WAR). These problems are addressed in the following manner:

RAW & WAW – A register locking mechanism is implemented in the register bank without the CU having to keep track of the locked registers. The acknowledgement signals, ZMs and ZAs, are asserted after the locking operation, and are de-asserted once the result is written back signalling the unlocking of the register. This implies that the destination register of the previous instruction will have been locked before the next one attempts to use that register. The timing diagram in Figure 5–12 assumes that the LD and ALU instructions write to different registers. Should the destinations be the same, then the **ZAs** acknowledgement signal would only be asserted after the **ZMs** acknowledgement signal has been de-asserted.

WAR – This hazard is avoided without additional hardware overheads. By definition, an instruction is issued when all of the acknowledgements from the relevant micro-operations have been received. This implies that the source registers of previous instructions will have already been accessed. Also, as long as the control signals to lock registers are not asserted before the operand fetch ones, then the register bank will ensure correct operation.

A behavioural description of the CU issuing a LDA instruction in this refinement step is given in Figure 5–3.

CASE instruction
.
.
LDA : SEQ

wait until ($\overline{RofA}$. $\overline{RyA}$. $\overline{MU3A}$. $\overline{ZMsA}$);	Wait until the previous handshakes on these control signals have been completed, by testing the input acknowledgement signals.	STATE 1.
assert (Rof, Ry, MU3, ZMs);	Initiate instruction execution.	
wait until (RofA . RyA . MU3A . ZMsA);	Wait until the Control Signals have been acknowledged.	STATE 2.
deassert (Rof, Ry, MU3, ZMs);	Instruction issued.	

Incoming Offset Register Acknowledgement Signal - RofA (Active phase), $\overline{RofA}$ (reset phase).
Outgoing Register Offset Control Signal - Rof

Figure 5–3: Issuing an LDA instruction in Refinement Step 2

Performance Results

Instruction	ICT	MML	Max. FM Utilisation
ALU	24nS	24nS	16.67%
LD	43nS	43nS	53.49%
ST	23nS	21nS	42.85%

Table 5–4: Instruction execution for Refinement Step 2

Test Programs	Alu Test	Load Test	Store Test	Hennessy Tests
Program Execution Time	175nS	308nS	164nS	106nS
MIICT	24nS	43nS	21nS	17nS
ALU Utilisation	16.57%	0%	0%	12%
MU Utilisation	0%	53.31%	39.87%	32%
Register Bank Utilisation	41.42%	23.18%	22.15%	39%
ALU Interface Utilisation	78.7%	0%	0%	57%
MU Interface Utilisation	0%	88.08%	82.91%	55%
Register Interface Util.	70.41%	92.72%	48.73%	71%

Table 5–5: Execution of the test programs on Refinement Step 2

This refinement step exploits limited spatial concurrency between instructions with different micro-operations, i.e. instructions which require different microagents. Therefore, improvements are only observed in the Hennessy Tests where instructions using different micro-operations (ALU and memory instructions) may execute concurrently, and this is reflected in the greater utilisation figures for the respective units as shown in Table 5–5.

5.10 Refinement Step 3 – Using a Single Write-back Bus

In the previous versions of the architecture, each functional unit had its own write-back bus which allowed result operands to be written back to the registers as soon as they became available. However, supporting *n* function units would require *n* write-back buses (incurring area costs) and *n* write-ports on the register bank (incurring performance costs). The micronet datapath (Figure 5–13) in this refinement step has only one write-back bus, i.e. the functional units share the ZM bus to write data back to the registers. The control signal **ZAs** is no longer used so there is only one write-back microagent control signal **ZMs**. This has a significant effect on performance since previous concurrent write-backs must now take place sequentially. Also, the instruction issue conditions forces instructions which require to write data back to execute completely sequentially again.

Performance Results

Instruction	ICT	MML	Max. FM Utilisation
ALU	24nS	24nS	16.67%
LD	43nS	43nS	53.49%
ST	23nS	21nS	42.85%

Table 5–6: Instruction execution on Refinement Step 3

Table 5–7 shows increases in the execution time for both Hennessy Test programs, which re-enforces the advantages of multiple write-back buses (see Table 5–5). Another interesting point to note is that the execution time of this test program is independent of data dependencies. Each instruction issue is

Test Programs	Alu Test	Load Test	Store Test	Hennessy Tests
Program Execution Time	175nS	308nS	164nS	139nS
MIICT	24nS	43nS	21nS	17nS
ALU Utilisation	16.57%	0%	0%	9.02%
MU Utilisation	0%	53.31%	39.87%	24.06%
Register Bank Utilisation	41.42%	23.18%	22.15%	33.83%
ALU Interface Utilisation	78.7%	0%	0%	42.86%
MU Interface Utilisation	0%	88.08%	82.91%	41.35%
Register Interface Util.	78.7%	44.04%	55.06%	66.17%
WB Bus Utilisation	37.28%	20.86%	39.87%	33.83%

Table 5–7: Execution of the test programs on Refinement Step 3

stalled until the previous one has written its result back to the registers. This is a return to almost complete sequential execution (as in Refinement Step 1). (The difference in PETs for the Hennessy Tests in Step 1 and here is due to concurrency between the ST and ALU operations.) Although the write-back bus doesn't seem to be a bottleneck, there are times when a result is delayed waiting for another write-back operation to complete. This can affect performance especially if the stalled data item is required by an instruction on the program's critical path.

5.11 Refinement Step 4 – Asynchronous Micro-operation Issue

In previous refinement steps, the control unit would not assert any of the individual control signals for issuing an instruction until all of them could be asserted together. This constraint is now relaxed so that once an instruction has been chosen to be issued, the individual control signals required by that instruction can be asserted asynchronously as soon possible. This allows micro-operations belonging to different instructions to overlap (see the timing diagram of Figure 5–14). Note that an instruction's control signals can only be de-asserted once all the relevant control signals have been acknowledged, this being the time at which the instruction is considered to have been issued (also shown in the timing diagram). This refinement aims to improve the instruction execution by exploiting a finer grain of ILP than previously possible in synchronous designs, i.e. concurrency between individual components within stages of a datapath. This also speeds up the instruction issue of blocked or stalled instructions. Only the control signals to the common resources (which have not finished) will be stalled thus allowing the ready resources to execute their micro-operations for the current instruction earlier than before. However, relaxing this constraint re-introduces possible hazards and efficient mechanisms have to be devised to avoid them.

Instruction Issue

The micro-operations for an instruction are initiated individually as soon as possible by asserting the necessary control signals. The receipt of an acknowledgement confirms that the associated micro-operation has begun and the instruction is said to be issued once all of the asserted control signals have been acknowledged. The initiating control signals can then be de-asserted and the next instruction issue can begin. As in Refinement Step 2, micro-operations relating to different instructions may overlap. However, while Step 2 benefited from spatial concurrency (made possible through the availability of resources), this refinement step exploits mainly temporal concurrency through a limited amount of pipelining. Fortunately, thanks to the properties of the micronet the hazard avoidance mechanisms are implicit in the orderings of the assertions of the control signals, known as *pre-issue conditions*, and these are discussed below. Since some micro-operations share the same resources they obviously cannot execute simultaneously. These restrictions are also applied by the pre-issue conditions.

RAW – An instruction is considered issued once all of its resource control signals have been acknowledged by the relevant microagents (i.e. the microagents are active). This allows the control signals to be cleared and the next instruction issue to begin. Recall that the control acknowledgement signal, **ZMs**, is asserted once the register is locked and cleared once data has been written to it. Thus, the destination register will be locked before the following instruction attempts to read from it, since the next instruction issue cannot be initiated until the previous set of control signals have been acknowledged.

WAR – When a register is used both as a source and a destination within the same instruction, then it is necessary to ensure that the source data is obtained before the register is locked, otherwise deadlock will occur. In the previous refinement steps no action was required to avoid this hazard since this criteria was met by the issuing conditions (the set of microagent control signals being asserted together) and the register bank. However, it is now possible for **ZMs** to be asserted before the source operand control signals **Rx** and **Ry** and therefore the CU stalls the assertion of **ZMs** until **Rx** and **Ry** have been asserted.

Operand fetch – It is also necessary to ensure that a functional unit gets the correct operands since it is possible for two units to require operands at the same time. Functional units fetch each of their operands separately over the operand fetch buses (X and Y) while acknowledging the control signal (i.e. operation request) from CU in the following manner:

1. If the bus is free and no other request is in progress then the request signal (to register port for this bus) is asserted.

2. When valid data is detected, the data is latched and the request signal is cleared. Data is, of course, only latched by the functional unit interface which made the original request.

Simultaneous operand requests by FMs to the same register bank CM micro-operation can lead to one of them acquiring the wrong operand. This can be avoided by the CU delaying the assertion of the control signal to one of the functional units. The CU need only delay the assertion of the control signal to a FM until the FM of the previous instruction has made its operand request(s) to the registers. This event will have occurred before the acknowledgement signals of the previous instruction's "operand fetch" micro-operations (**Rx**, **Ry** or **Rz**) have been de-asserted.

WAW – A situation may arise where the current instruction is stalled because a previous instruction has not written its result back to the destination register. This stall is necessary because the current instruction might either attempt to write its result to an unlocked register (which may eventually cause a deadlock) or write data to a location out of program order. In this refinement, write-backs are still forced to occur in-order. The solution adopted here is very simple since the above conditions can be avoided by preventing each functional unit from writing data back until its control signal from the CU has been de-asserted (an implicit go-write signal). This is sufficient since an instruction's control signals cannot be de-asserted before **ZMs** is asserted (see timing diagram in Figure 5–14). (In the CU, the control signals will be de-asserted once all the required acknowledgements have been received, which includes **ZMs**, implying that the destination register has been locked.) Note that if the CU attempts to lock a register which is already so, then the acknowledgement signal will not be asserted and the current register lock request will stall. This mechanism guarantees that write-backs to the same register occur in the correct order without stalling the instruction issue, and thereby allowing the instructions to execute concurrently with only write-backs taking place sequentially. Historically, the CDC6600 [162] used a *Go-Write* signal which sequentialised the execution of the offending instructions.

Bus Contention – Only the functional units and the register bank can write on to the Z Bus. The mechanism to avoid WAW hazards prevents contention between functional units and therefore the only possibility for contention is when the Register Bank and one of the functional units attempt to write on the bus simultaneously. However, access to this bus is arbitrated by the CU, through the mutually-exclusive assertions of the operand fetch control signal **Rz**, and the write-back control signal **ZMs**.

The refinements to the behavioural description of the CU issuing a LDA instruction are shown in Figure 5–4.

CASE instruction
    .
    .
  LDA : SEQ
      PAR
        wait until ($\overline{\text{RofA}}$. $\overline{\text{RxA}}$); assert (Rof);
        wait until ($\overline{\text{RyA}}$); assert (Ry);
        wait until ($\overline{\text{RofA}}$. $\overline{\text{RxA}}$. $\overline{\text{RyA}}$. $\overline{\text{ZMsA}}$); assert (ZMs);
        wait until ($\overline{\text{RofA}}$. $\overline{\text{RxA}}$. $\overline{\text{RyA}}$. $\overline{\text{MU3A}}$); assert (MU3);

Assert Control Signals in accordance with the pre-issue conditions.

      PAR
        wait until (RofA); deassert (Rof);
        wait until (RyA); deassert (Ry);
        wait until (ZMsA); deassert (ZMs);
        wait until (MU3A); deassert (MU3);

Wait for Control Signal acknowledgements.

Incoming Offset Register Acknowledgement Signal	- RofA (Active phase), $\overline{\text{RofA}}$ (reset phase).
Outgoing Register Offset Control Signal	- Rof

Figure 5–4: Issuing an LDA instruction in Refinement Step 4

Performance Results

Instruction	ICT	MML	Max. FM Utilisation
ALU	21nS	24nS	19.05%
LD	42nS	43nS	54.76%
ST	23nS	21nS	42.85%

Table 5–8: Instruction execution on Refinement Step 4

Test Programs	Alu Test	Load Test	Store Test	Hennessy Tests
Program Execution Time	157nS	302nS	165nS	119nS
MIICT	21nS	42nS	22nS	16nS
ALU Utilisation	18.54%	0%	0%	10.62%
MU Utilisation	0%	54.39%	39.62%	28.32%
Register Bank Utilisation	37.09%	23.65%	35.85%	43.36%
ALU Interface Utilisation	80.13%	0%	0%	68.14%
MU Interface Utilisation	0%	89.86%	78.62%	48.67%
Register Interface Util.	93.38%	89.19%	88.68%	77.88%

Table 5–9: Execution of the test programs on Refinement Step 4

Table 5–9 shows some improvement in the execution times over Refinement Step 3. In fact the PETs for the instruction test programs are better than the corresponding values in Refinement Step 2 (see Table 5–5). These performance gains are due to the small improvements in the instruction cycle times as shown in Table 5–8. The magnitude is determined by the overlap between the operand access of the current instruction and the write-back of the previous one. In the example under consideration there can only be two program instructions active in the datapath simultaneously. The likelihood of operand fetches and write-backs occurring concurrently depends on the FM delay.

Although the Hennessy Test PETs also show improvements over the previous refinement step, they are still worse then the figures in Step 2. In Refinement Step 2, the programs exploited spatial parallelism, whereas now they only exploit temporal parallelism. The latter is limited, due to the control unit being unable to complete the issuing of the current instruction, (specifically, locking the destination register) until the previous instruction has written its result back to the register. This is necessary to enforce in-order instruction completion and to prevent contention on the write-back bus. Also, the MIICT for the Hennessy Test (in Figure 5–9) is less than the corresponding values for the other test programs. This is due to the overlapping of independent instruction issues.

While analytical estimates of program execution times (PETs) for the Alu, Load and Store Tests (see Equation 5.1) match those obtained from the simulation, it is less easy to obtain the same for programs with a mix of instructions, as in the case of the Hennessy Test. The execution times for such programs depend on a number of factors, such as the relative values of the instruction issue and cycle times and resource availability, which affect the amount of concurrency available.

5.12 Refinement Step 5 – Out-of-Order Write-Backs

Enforcing in-order write-backs restricts the amount of concurrency which can be exploited especially when functional unit execution times vary significantly. However, supporting out-of-order completion of instructions in an asynchronous environment is more difficult than under synchronous control. Determining the precise order in which results will become available is virtually impossible since micro-operation delays vary (subject to data and environmental parameters). Therefore a decentralised bus arbitration scheme is required such as a token ring which is distributed amongst the CMs that write to the bus. Out-of-order instruction completion can now be supported by tagging the write-back data with the address of its destination register. However, the micro-operation to write data back to the register bank can no longer be controlled by the CU since the order of the write-backs cannot be predicted. Therefore, write-backs are initiated directly by the CMs of the FMs which require the service, i.e. the write-back micro-operation is initiated by the micro-operations in the previous stage.

Since the Z bus is shared by the functional units which generate results and access to the bus is no longer controlled by the CU, then potential for bus contention does exist. Two (or more) CMs may attempt to write on to the bus at the same time (or within the bus propagation delay). Determining the precise times of the availability of data is very difficult. The use of a centralised request-grant arbitration scheme is possible. This will become more complex as the number of functional units increases. A priority scheme could be incorporated to give certain functional units, especially those with longer delays like the memory unit, access to the bus before other waiting units. An alternative more distributed scheme can be achieved by using a token ring. The token need only be held for the duration of data transfer and not the whole handshake. The ring is distributed amongst the FU interfaces and is very simple to implement. However as the number of functional units increases, so does the token's cycle time and for architectures with a large number of FUs this may not prove to be a satisfactory solution.

The register control signal **ZMs** has to be modified in order to *decouple* the CU from the process of writing data back into the register:

ZMs – Now just locks the destination register and prevents read access to it. The corresponding control signal acknowledgement is now set on receiving the request (the asserted **ZMs** control signal) from the CU, and cleared when the register is locked. Again, **ZMs** and **Rz** cannot be asserted simultaneously, since it is now necessary to guarantee that either the register has been locked prior to the next instruction being issued (in case of a RAW dependency), or that the register has been read before it is locked (in the case of a WAR dependency). Note that in the case of a WAW dependency, it is still necessary for the functional unit control signal to be de-asserted after the destination register has been locked, i.e. de-asserted only after the de-assertion of **ZMs** has been acknowledged.

Performance Results

Instruction	Instruction Cycle Time	Micronet Latency	Maximum FU Utilisation
ALU	21nS	24nS	19.05%
LD	42nS	43nS	54.76%
ST	23nS	21nS	42.85%

Table 5–10: Instruction execution for Refinement Step 5

Test Programs	Alu Test	Load Test	Store Test	Hennessy Tests
Program Execution Time	159nS	302nS	165nS	114nS
MIICT	21nS	42nS	23nS	17nS
ALU Utilisation	18.3%	0%	0%	11.11%
MU Utilisation	0%	54.39%	39.62%	29.63%
Register Bank Utilisation	29.41%	23.65%	32.70%	56.48%
ALU Interface Utilisation	80.39%	0%	0%	72.22%
MU Interface Utilisation	0%	89.86%	79.25%	72.22%
Register Interface Util.	86.27%	85.81%	91.19%	85.19%

Table 5–11: Execution of the test programs on Refinement Step 5

The results in Table 5–11 show that in this refinement, out-of-order instruction completions (i.e. write-backs) have little effect on performance. This is to be expected in the instruction test programs where there is no scope at all for benefit, although the Hennessy Test shows is only a slight improvement. The explanation is as follows: In order to benefit from out-of-order write-backs, the architecture needs to be able to exploit spatial parallelism. In the micronet, this means that the instruction issue rate needs to be faster than the instruction execution rates. It can be observed in Table 5–11, that the Minimum Instruction Issue Cycle Time (MIICT) is nearly as long as the smallest Instruction Cycle Time (ICT). This suggests that the issue of instructions is a limiting factor on the degree of spatial concurrency that can be exploited. In order to achieve higher concurrency it is necessary for the IICT to be as small a proportion of the smallest ICT as possible. Another reason is the limited amount of spatial parallelism available in the test programs themselves and the general (conservative) dependency rules applied when issuing instructions. These issues will be addressed in following refinements.

5.13 Refinement Step 6 – Faster Instruction Issue

The issue cycle time determines the rate at which instructions can be issued to the micronet datapath and should this be a limiting factor on performance then the handshake cycle times of the microagent control signals have to be minimised. This can be achieved by either improving the hardware design of the control circuits, or alternatively, by redefining the handshake cycle itself (the option considered in this refinement step).

Here, in Refinement Step 6, the rôle of the CU is diminished further by distributing the control of the micronet to individual CMs. While the CU initiates the micro-operations individually for the current instruction as early as possible via the corresponding CMs as before, the rôle of the CMs has been enhanced to more than just controlling local communications between FMs. They effectively buffer the initiation of the micro-operations from the CU until the respective FMs are ready to perform. This increases the number of operations which actually take place concurrently. This is also due in part to the changes in the significance of the control signal handshake. The acknowledgements to the control signals are revised as shown below:

Rx – This signal still identifies the source register whose contents are to be transferred across the X Bus. However, the corresponding acknowledgement is asserted by the CM of the register bank when the X bus operand fetch micro-operation is *ready* to access the register, and de-asserted once the operand fetch handshake is in progress over the X bus.

Ry – Same as above, but for the Y Bus.

Rz – Same as above, but for the Z Bus.

Rof – Same as above, but also with the restriction that the control signals **Rx** and **Rof** cannot both be active simultaneously.

Rz – The acknowledgement signal is cleared when the register interface has received the data transfer acknowledgement from the destination functional unit. (Z bus is data driven).

AUs – This still identifies the next operation to be carried out by the ALU. The acknowledgement, however, is now asserted when the corresponding CMs are ready to fetch the operands from the registers and is cleared once the FM micro-operation has completed.

MC1 – This signal still identifies a load instruction for the MU. The acknowledgement is asserted and cleared as for **AUs**.

ZMs – This signal still identifies the destination register which has to be locked. However, the corresponding acknowledgement signal is asserted when the CM is ready and de-asserted once the operation has been completed, as described in the previous refinement step.

As in previous refinement steps, hazards are dealt with by properly sequencing the control signals (the pre-issue conditions):

WAR – A functional unit cannot generate a result without first receiving its input operands. These are fetched in instruction order due to the handshake mechanism. The **ZMs** signal is only asserted after all the previous operand fetch control signal handshakes have been *completed*. This also prevents the destination register being locked before operands are accessed.

WAW – The mechanism is similar to before, except now the de-assertion of the functional unit control signals is no longer delayed until the **ZMs** acknowledgement signal is de-asserted. Instead, the go-write signal now originates explicitly from the register interface once the register has been locked and not implicitly from the CU.

RAW – The CU delays the assertion of the operand fetch control signals **Rx**, **Ry** and **Rz** until the previous **ZMs** control acknowledgement signal has been de-asserted, which indicates the locking of the previous destination register.

Operand Fetch – The pre-issue conditions are same as before. For each instruction, the control signal to the functional unit interface is only asserted after the required operand fetch control signals. This prevents bus contention on the operand fetch buses and guarantees that operands will be fetched in-order.

Write-back Contention – This is prevented by the use of a token ring to arbitrate accesses to the write-back (Z) bus. Of course, this problem could be obviated by using dedicated buses for small number of FMs, but may be impractical for larger designs.

The behavioural description of the CU issuing a LDA instruction in this refinement step is given in Figure 5–5.

CASE instruction
    .
    .
    LDA : SEQ
        PAR
          wait until ($\overline{\text{RofA}}$. $\overline{\text{RxA}}$. $\overline{\text{ZMsA}}$); assert (Rof);
          wait until ($\overline{\text{RyA}}$. $\overline{\text{ZMsA}}$); assert (Ry);
          wait until ($\overline{\text{RofA}}$. $\overline{\text{RxA}}$. $\overline{\text{RyA}}$. $\overline{\text{ZMsA}}$); assert (ZMs);
          wait until ($\overline{\text{RofA}}$. $\overline{\text{RxA}}$. $\overline{\text{RyA}}$. $\overline{\text{MU3A}}$); assert (MU3);
        PAR
          wait until (RofA); deassert (Rof);
          wait until (RyA); deassert (Ry);
          wait until (ZMsA); deassert (ZMs);
          wait until (MU3A); deassert (MU3);

Assert Control Signals in accordance with the pre-issue conditions.

Wait for Control Signal acknowledgements.

Incoming Offset Register Acknowledgement Signal - RofA (Active phase), $\overline{\text{RofA}}$ (reset phase).
Outgoing Register Offset Control Signal - Rof

Figure 5–5: Issuing an LDA instruction in Refinement Step 6

Performance Results

Instruction	ICT	MML	Max. FM Utilisation
ALU	15nS	24nS	26.67%
LD	39nS	43nS	58.97%
ST	23nS	21nS	42.85%

Table 5–12: Instruction execution on Refinement Step 6

Test Programs	Alu Test	Load Test	Store Test	Hennessy Tests
Program Execution Time	123nS	287nS	165nS	102nS
MIICT	15nS	39nS	23nS	10nS
ALU Utilisation	23.93%	0%	0%	12.5%
MU Utilisation	0%	57.3%	39.62%	33.33%
Register Bank Utilisation	59.83%	24.91%	35.22%	66.67%
ALU Interface Utilisation	94.87%	0%	0%	83.33%
MU Interface Utilisation	0%	97.86%	79.25%	84.38%
Register Interface Util.	93.16%	89.32%	93.08%	90.63%

Table 5–13: Execution of the test programs on Refinement Step 6

The instruction issue times have been reduced by minimising the delay between the assertion of the control signals and the arrival of their acknowledgements. This improvement was achieved by pipelining the control signal handshakes. Previously, the control acknowledgement signals were asserted once the particular action had taken place. Now, the interfaces (if not already busy with a previous handshake) will acknowledge a request immediately, signifying that the operation will take place, and de-assert the acknowledgement signal once the task has been completed. Although the particular task may have been completed, the interface may still continue to be busy completing successive tasks and may not be ready to acknowledge another request from the CU immediately, thus hiding communication delays.

An improvement in the ICTs of instructions which require to write data back to the registers, i.e. the LD and ALU instructions, can be observed in Table 5–12. This is due to the de-centralisation of the write-back control to the relevant CMs. These improvements are reflected in the shorter PETs for the Alu, Load, HT1 and HT2 test programs, as shown in Table 5–13. Thus, the faster issue cycle time allows these test programs to benefit from the write-back modifications made in the earlier refinement steps.

5.14 Refinement Step 7 – Data Forwarding

This refinement step implements two features to increase the amount of fine-grained concurrency which is available: firstly, the well-known technique of data forwarding to reduce the effect of stalls due to data dependencies between instructions and secondly, the application of the pre-issue conditions only when strictly necessary in order to reduce CU stalls. Implementing the specific dependency rules requires checking the actual register addresses within instructions. This requires extra hardware and increases the control unit's complexity, however all of this is required for out-of-order instruction issue. Therefore, it may only be worthwhile if the expected or exploitable performance (determined by the target application) outweighs the cost of implementing the specific dependency rules (which depends on implementation technology). Alternatively, it may be possible to generate the required information at compile time and encode it into the instruction word.

Previously, the micronet imposed a feed-forward discipline in the pipelines. This is now relaxed by having feedback paths which has the effect of allowing required operands to move against the flow. In the micronet datapath, the operands are only available for a short period of time (i.e. while they are being transferred to the register interface) after which they are obtained from the register bank (in effect the architecture implements only one stage of the counterflow pipeline [157], and relies on fast operand fetch access from the register bank).

With data transfer on the Z bus being tagged, the CMs can identify and intercept operands for which they may be waiting. This mechanism, reminiscent of the IBM 360/91 common bus architecture [163], has been implemented by exploiting the feedback loops within the micronet. In the event of data forwarding, where data is routed directly to the CM of the waiting FM, the CM's previous request for that operand is, in effect, cancelled by initiating a separate handshake. This frees the corresponding operand fetch CM to service its next request. An alternative approach would be to implement operand bypassing, where the operand is fed back to the operand fetch micro-operation. This feature avoids both duplicated tag matching in each of the data forwarding CMs and the need for the cancel handshake, at the expense of being slower than data forwarding. However, the functionality is viewed as internal to the register bank, since, from outside the data is obtained from the same place – just slightly quicker than expected. This means that this method would fit perfectly into the micronet model since no further modification would be required to any other part of the datapath.

The dual rôle of the Z Bus can now no longer be supported due to the data-forwarding mechanism. A separate operand fetch bus (W Bus) is used, making the Z Bus purely a write-back bus (see Figure 5–17). (In the previous refinements, the Z Bus was used as both a operand bus (for STR instructions) and as a write-back bus.) By separating the functionality, the register interface for the Z bus is simplified. The traffic on the Z Bus is reduced and since the register interface no longer needs to send data on the Z Bus, it can be removed from the token ring (speeding up the ring's cycle time). Also, this allows the third operand for a STR instruction to be forwarded when necessary. As one might expect, in terms of exploiting concurrency, it is better if less resources are shared between operations of different pipeline stages.

Performance Results

Columns "HT2" and "HT2$_{DF}$" refer to the cases without and with data-

Instruction	ICT	MML	Max. FM Utilisation
ALU	15nS	24nS	26.66%
LD	38nS	43nS	60.52%
ST	23nS	21nS	42.85%

Table 5–14: Instruction execution on Refinement Step 7

Test Programs	Alu Test	Load Test	Store Test	HT1	HT2	$HT2_{DF}$
PET	121nS	280nS	165nS	83nS	97nS	91nS
MIICT	15nS	39nS	23nS	8nS	10nS	10nS
ALU Util.	24.35%	0%	0%	15.58%	13.18%	14.11%
MU Util.	0%	58.76%	39.62%	41.55%	35.16%	37.65%
Reg. Bank Util.	60.87%	25.55%	22.01%	58.44%	67.03%	65.88%
ALU If. Util.	94.78%	0%	0%	79.22%	80.21%	80%
MU If. Util.	0%	97.81%	79.25%	72.72%	74.72%	72.94%
Reg. If. Util.	94.78%	89.42%	93.08%	90.90%	92.30%	92.94%

Table 5–15: Execution of the test programs on Refinement Step 7

forwarding, respectively. As is expected, the results show improvements in performance in programs with data dependent instructions, and this is recorded in the figures for the Hennessy Test in Table 5–15. As a side-effect of the data-forwarding, the PET improvements in the Load and Alu Test are due to the introduction of the W Bus which removed the Register Interface from the token ring, thereby reducing the ring's cycle time. The division of the Z bus into separate buses also improves the scope for greater concurrency. For the first time the PETs for HT1 and HT2 differ since the pre-issue conditions have been applied only when necessary. Since HT1 has no data dependencies between instructions it executes faster.

5.15 Refinement Step 8 – The Last Control Modification

In this final refinement step, both the assertion and de-assertion of the control signals occur independently of each other. This increases further the concurrency between micro-operations and maximises the exploitation of fine-grained concurrency between instructions for a given architecture. A behavioural description of the CU issuing a LDA instruction is given in Figure 5–6. Previously, the FM control signal acknowledgements represented the business of their respective functional units. This is no longer the case, since these signals are de-asserted on the receipt of the required operands. This effectively reduces the ICT as is observed in the performance figures for all the programs in Table 5–17.

PAR

wait until ($\overline{\mathrm{RofA}}$. $\overline{\mathrm{RxA}}$. $\overline{\mathit{ZMsA}}$); assert (Rof); wait until (RofA); deassert (Rof);

wait until ($\overline{\mathrm{RyA}}$. $\overline{\mathit{ZMsA}}$); assert (Ry); wait until (RyA); deassert (Ry);

wait until ($\overline{\mathit{RofA}}$. $\overline{\mathit{RxA}}$. $\overline{\mathit{RyA}}$. $\overline{\mathrm{ZMsA}}$); assert (ZMs); wait until (ZMsA); deassert (ZMs);

wait until ($\overline{\mathrm{RofA}}$. $\overline{\mathrm{RxA}}$. $\overline{\mathrm{RyA}}$. $\overline{\mathrm{ZMsA}}$. $\overline{\mathrm{MU3A}}$); assert (MU3); wait until (MU3A); deassert (MU3);

Incoming Offset Register Acknowledgement Signal	-	RofA (Active phase), $\overline{\mathrm{RofA}}$ (reset phase).
Outgoing Register Offset Control Signal	-	Rof
Condition only applied when a dependency exists	-	*RofA*

Figure 5–6: Issuing an LDA instruction in Refinement Step 8

Performance Results

The ICT figure for the LD instruction in Refinement Step 8 is the best attainable as it represents the MU's FM delay for the operation. The corresponding

Instruction	ICT	MML	Max. FM Utilisation
ALU	12nS	24nS	33.33%
LD	23nS	43nS	100%
ST	12nS	21nS	75%

Table 5–16: Instruction execution for Refinement Step 8

Test Programs	Alu Test	Load Test	Store Test	HT1	$HT2_{DF}$
Program Exec Time	103nS	188nS	98nS	79nS	91nS
Effective Speed Up	1.75	1.66	1.71	1.89	1.62
MIICT	10nS	10nS	10nS	10nS	10nS
ALU Utilisation	28.87%	0%	0%	16.44%	14.11%
MU Utilisation	0%	88.46%	67.74%	43.84%	37.65%
Register Bank Util.	72.16%	32.45%	37.63%	64.38%	64.71%
ALU Interface Util.	93.81%	0%	0%	78.08%	78.82%
MU Interface Util.	0%	96.7%	95.7%	72.6%	70.59%
Register Interface Util.	93.81%	79.79%	90.32%	91.78%	91.76%

Table 5–17: Execution of the test programs on Refinement Step 8

utilisation figure in Table 5–17 supports this claim (Note: these utilisation measurements do take into account both the initial operand fetch and the final write-back delays, and will therefore never attain the theoretical upper bound shown in Table 5–16). These figures show that the micronet can exploit the actual operational costs and effectively hide the overheads of self-timed design. The ICTs for the ALU and ST instructions are limited by their operand fetch cycle times, and the utilisation of the FM in these cases also approach their bounds. These cycle times are due to the communication protocol between the FUs and the register bank. These delays can be reduced by using a less conservative bundling delay [158] and through layout and transistor size optimisation [26]

(Refinement Step 9). The improvements in FM utilisation over the 9 refinement steps are shown in Figure 5–7.

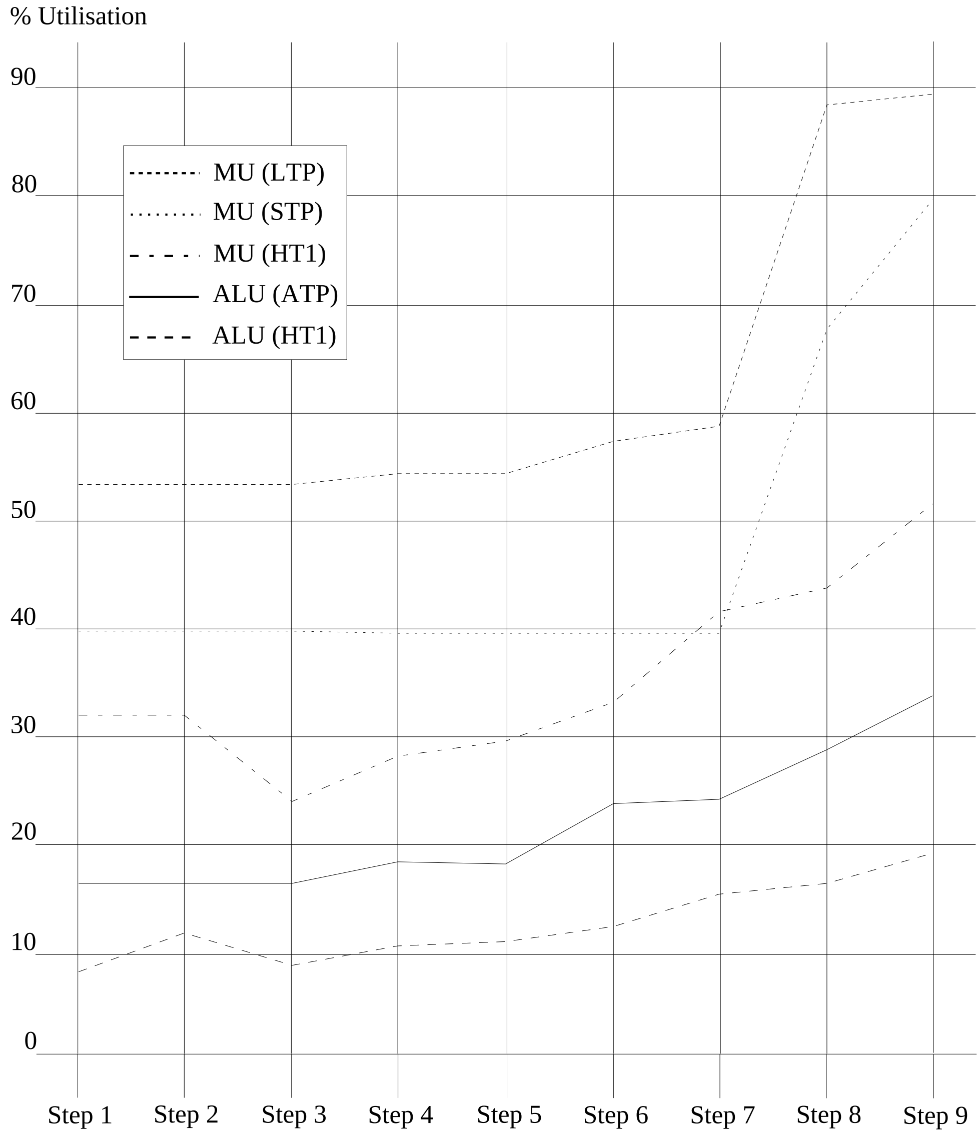


Figure 5–7: The FM utilisations

The overall improvements in the program execution times in Refinement Step 8 over Step 1 (shown in Table 5–17 and Figure 5–8) for the three instruction test programs are due to improvements in temporal concurrency due to asynchronous pipelining of the datapath. The actual speedups achieved are less than the maximum attainable improvement, which is the ratio of the ICTs (in

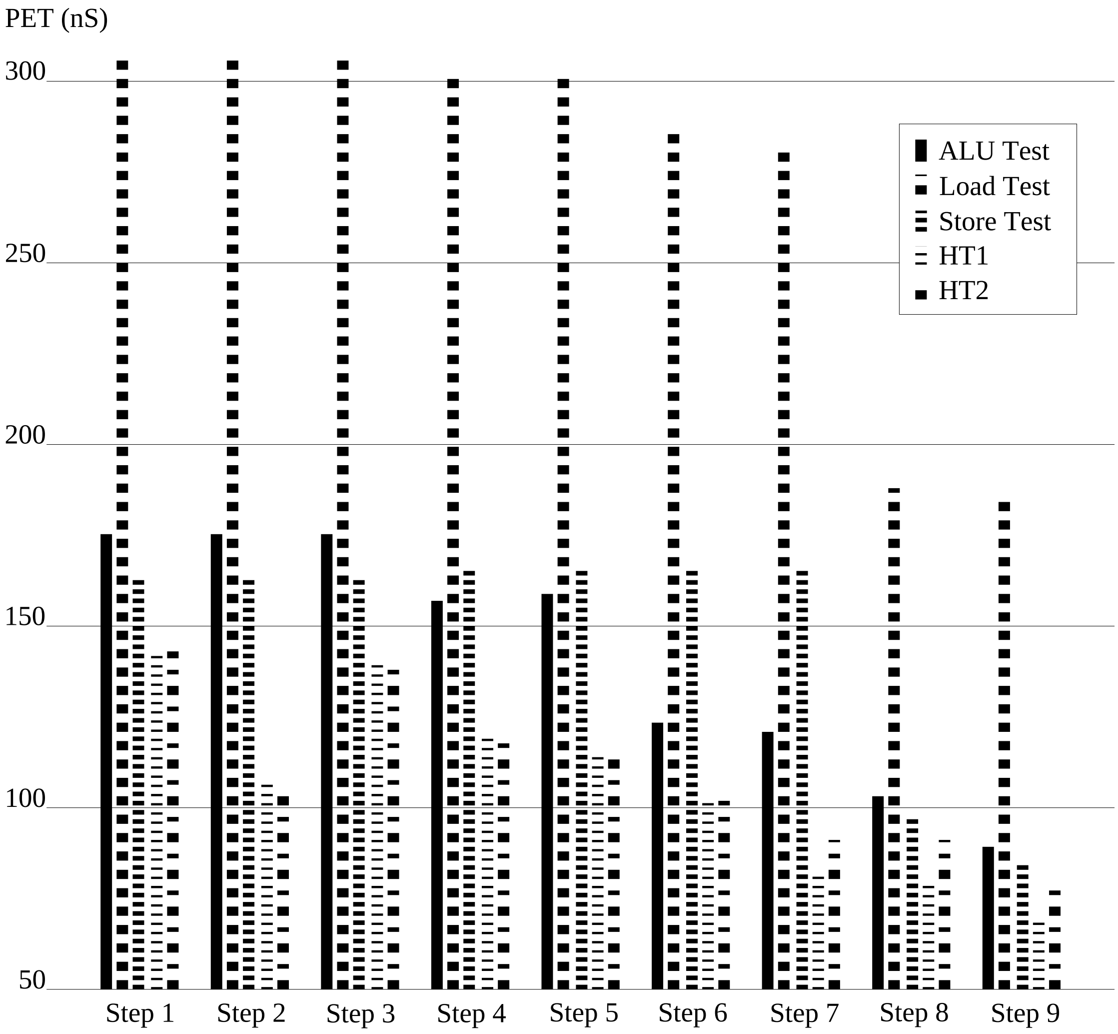


Figure 5–8: The test program execution times

Tables 5–2 and 5–16), because of the MML and the startup overheads (see Equation 5.1), for longer tests programs the speed-up will approach the maximum value. The speed-up for HT1 is in part due to the pipelining of the instructions as observed in previous test programs, and also due to additional spatial concurrency achieved through the overlapping of different instructions in the same stage of the micronet. This further improvement is still significant (approximately 17-20% in this example) given that both successive instruction operand fetches and write-backs are effectively forced to take place sequentially due to resource constraints. (In fact, the delays of these operations are larger than the

FM delays for store and ALU operations and the MIICT which implies that the scope for spatial concurrency in this particular example is quite small). As the number of microagents within each stage is increased, the spatial concurrency effect will be more pronounced. The speed-up for HT2, as expected, reflects the reduced concurrency which can be exploited, because of data dependencies in the program. These dependencies affect spatial concurrency more since they sequentialise operations irrespective of resource requirements. This emphasises the need for a good instruction schedule to exploit micronet-based architectures.

5.16 Conclusions

The interaction between concurrently executing instructions is quite difficult to predict. For example, two instructions which compete for the same resources might acquire them in a different order depending on the actual delays which are themselves data-dependent. This is not in itself a drawback, since one of the instructions is stalled for just as long as is necessary, unlike the synchronous case.

These refinements have investigated the influence of an asynchronous control paradigm on the performance of processor architectures for exploiting fine-grained ILP. The rôle of the CU in an asynchronous processor can be considerably simplified, just to initiating individual micro-operations as early as possible. The control of the datapath is distributed to local interfaces courtesy of the micronet. The results show that given a set of resources, an asynchronous control paradigm implemented as a micronet is able to efficiently achieve good utilisation on datapath resources through the exploitation of both actual execution latencies and fine-grain spatial and temporal ILP. It has to be noted that the datapath latency is unaffected by the exploitation of temporal parallelism which is generally not the case in a synchronous pipeline.

5.17 Refinement Step 9 – Transistor Resizing

This additional refinement further illustrates how easily modifications can made to the behaviour of a particular part of the micronet without affecting the rest of the design. In the previous refinement step, some of the ICTs were limited by the handshake cycle time of data transfers across the buses. In this refinement step, the bus drivers have been resized to reduce the bus propagation times. Consequently, no other (design) modifications were required to both ensure the correct operation of the micronet and exploit the benefits.

Performance Results

Instruction	ICT	MML	Max. FM Utilisation
ALU	10nS	22nS	40%
LD	23nS	41nS	100%
ST	10nS	20nS	90%

Table 5–18: Instruction execution for Refinement Step 9

In this refinement step, the ICTs of both the ALU and ST instruction, as shown in Table 5–18, are now limited by the instruction issue cycle. This is clearly highlighted by the fact that their corresponding test programs have similar PETs (see Table 5–19).

In a synchronous pipeline, performance benefits can only be attained when improvements are made to the worst-case delay of the slowest stage. However, a micronet can exploit the benefits of improvements made to any stage. In MAP, the delay associated with the current issue cycle is determined by the current slowest stage and this is likely to vary from cycle to cycle. Excluding hazards, the instruction issue rate is limited by the issue cycle time or the operand fetch

Test Programs	Alu Test	Load Test	Store Test	HT1	HT2	$HT2_{DF}$
PET	89nS	186nS	85nS	68nS	83nS	78nS
MIICT	10nS	12nS	10nS	10nS	10nS	10nS
ALU Util.	33.75%	0%	0%	19.35%	15.58%	16.67%
MU Util.	0%	89.44%	79.75%	51.61%	41.56%	44.44%
Reg. Bank Util.	77.11%	36.67%	44.3%	62.9%	70.13%	68.06%
ALU If. Util.	92.77%	0%	0%	74.19%	75.32%	69.44%
MU If. Util.	0%	96.67%	96.2%	74.19%	74.03%	72.22%
Reg If. Util.	91.57%	81.11%	88.61%	90.32%	90.91%	91.67%

Table 5–19: Execution of the test programs on Refinement Step 9

cycle time (depending on the delays incurred by their actual implementation). The time to write data back to registers will vary depending on the time taken to obtain access to the bus.

5.18 Discussion

The work in this chapter has focused on how asynchronous controls within the micronet can be used efficiently to utilise the components of a typical datapath. The architecture has been influenced to the extent that resources (microagents) should operate as independently and concurrently as possible. This is achieved through the distribution and decentralisation of control and the use of decoupling queues between successive resources. The micronet effectively provides a framework to control a processor architecture and does not constrain a designer towards any particular architecture. Instead the designer can easily add or remove resources (thus modifying the architecture) to meet different design criteria. The processor's performance and efficiency is determined by the proportion of the actual delays (latencies) of the micro-operations to the cycle times of their functional units (FMs). It is difficult to quantify the magnitude of change a particular modification will have on performance because of the complex interdependency (sequencing and actual delays of components) of events. A high functional unit utilisation isn't necessarily a good thing since this may imply a bottleneck in that particular unit. Of course, the designer could replace that unit with a faster one or even add another unit to improve performance which may have the effect of reducing the utilisation. Improving the performance of the part of a design which is causing a bottleneck will simply move the bottleneck to the next slowest part of the design (which may or may not be within the design constraints). This work tries to advocate that given a set of architectural resources, the micronet control paradigm is better able utilise them (almost achieving their maximum theoretical bounds).

5.18.1 Minimising the Self-Timed Overheads

As shown in this chapter, the key to efficient exploitation of ILP in MAP has been the ability to hide the overhead due to asynchronous handshake protocols. While the two-phase handshaking protocol is conceptually easier to understand, a four-phase one leads to simpler and therefore possibly faster circuits. In order to exploit ILP, the control unit might want to issue instructions before the previous one has completed, therefore these circuits need to be as fast as possible.

Both transitions in a generic four-phase protocol (the assertion and the return-to-zero) are accompanied by additional acknowledgements from the receiver. Although the principal advantage of this approach is a simpler circuit implementation, it uses twice as many transitions than is necessary and whenever the wire delay is a substantial fraction of the operation time, the extra trip required by a single communication can be a serious performance penalty.

In fact, the reset phase of the handshake does not signal any event, thus leading some designers to modify the protocol to simultaneously reset the two signals after the active phase to reduce the handshake cycle time [55]. The micronet is only concerned with the external communications between microagents, each of which might use a different protocol internally. Micronets employ the traditional four-phase handshaking protocol for both control and local bundled data transfer. Other reasons, more specific to micronets, have influenced this choice, and these are discussed next.

Fast Instruction Issue

One of the significant features of micronets is their ability to exploit spatial concurrency within the datapath. This requires a fast instruction issue rate to keep the microagents busy. The CU initiates the micro-operations for each of the instructions individually and as early as possible. The acknowledgements from the CMs (after a delay of one C-element) confirm that the corresponding micro-operations will be initiated. The instruction is considered to have been issued once the CU has received all the acknowledgements. This corresponds to the first half of the four-phase protocol. The CU is free to issue the next instruction, while the reset phase of the protocol completes. This is done when the corresponding acknowledgement signal is de-asserted which signifies that the particular resource is ready for the next request. The instruction releases the resources individually as soon as the respective micro-operations have completed, thus freeing the resources for another instruction. Figure 5–9 shows the activity of two resources in micronets in comparison to a micropipeline and a synchronous pipeline. Assume each of the three instructions require two resources concurrently for varying periods of times. In the case of the synchronous pipeline, successive instructions must wait for the next clock tick to begin execution. In a micropipeline, the next instruction can begin execution when the previous one has finished with both resources. In both cases, significant

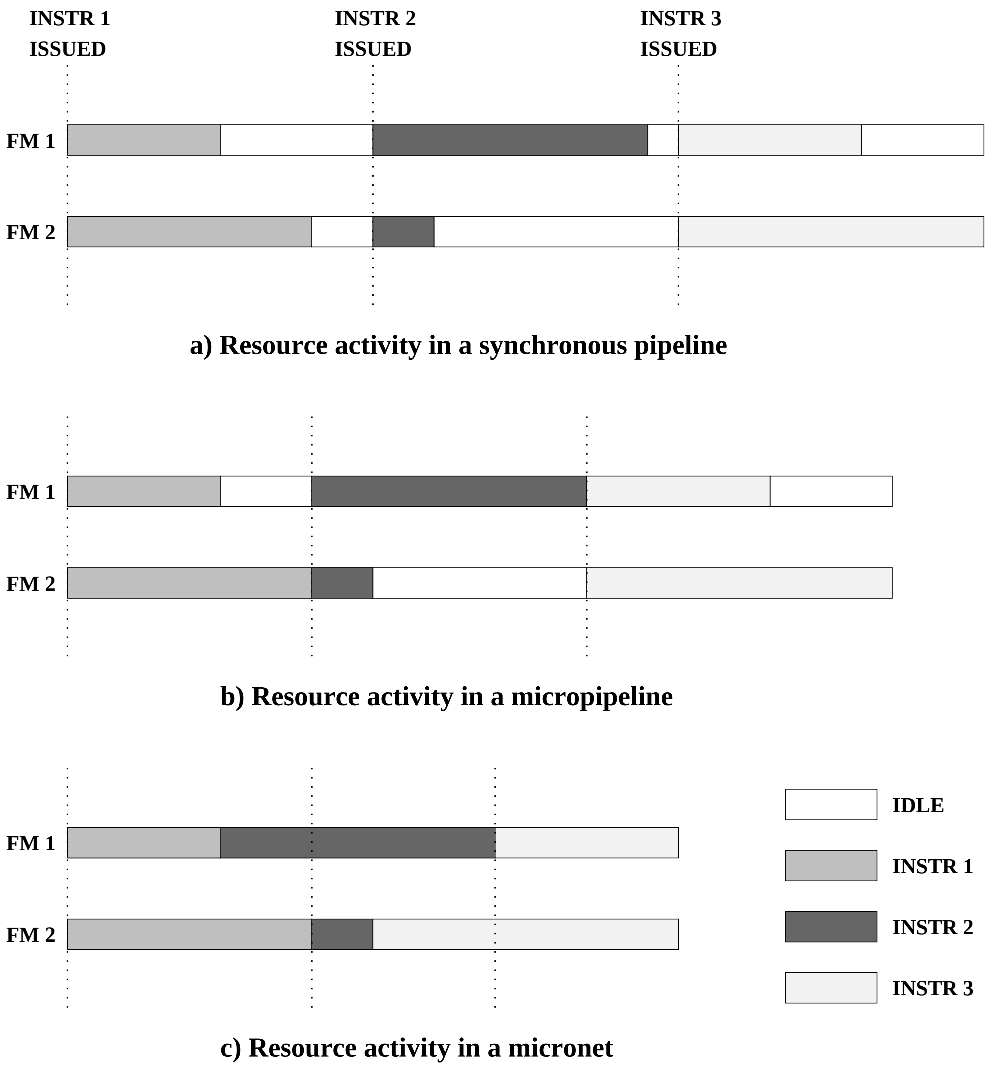


Figure 5–9: Resource activity

idle times may exist. A micronet can reduce these idle times by not forcing the instructions to obtain both resources at the same time.

Allowing the micro-operations of different instructions to overlap could lead to potential hazards. Since the acknowledgement signals effectively denote the readiness or busyness of resources, they can be collectively used as a scoreboard. Hazard avoidance due to data dependences is implicit in the orderings of the assertions of the control signals. These *pre-issue conditions* stall the assertion of the respective control signal until the completion of one of the halves of the handshake protocol of the dependent micro-operation control signal(s). Although a four-phase protocol would be considered twice as expensive as a two-phase one, the same efficiency is obtained as two back-to-back, two-phase handshakes by representing two events in each cycle. The recovery transitions are used by the control unit for scoreboarding and hazard avoidance. This is necessary for efficient exploitation of ILP, since the control unit has to issue each instruction before the previous one completes its execution. Furthermore, a four-phase protocol exposes more concurrency by effectively decoupling the sender's and receiver's operations from their communication.

Routing Data in Micronets

Although the actual data transfer between microagents is controlled locally via handshake protocols, the access to shared resources, such as data highways, may be controlled either globally by the CU or locally by an arbitration scheme. Global control is used in cases where the order of granting resources is known in advance or has to be enforced. This is again achieved through the use of pre-issue conditions. Otherwise, a local mutual exclusion scheme such as in token rings or arbiters will grant requests. For example, the writing back to the register bank is controlled directly by the CMs of the FMs which require this service. As a consequence of this and also due to the differences in the execution times of micro-operations, instructions may complete out of order. Therefore data has to be tagged with its destination which also enables data-forwarding to be supported.

The decision to use a two-phase or four-phase protocol also depends on whether the local communication between two microagents takes place over a shared bus or not. When wires are shared between two or more components, the wires must return to an inactive or predetermined state to allow successive handshakes to commence. When the highway is not shared, two-phase can be used because there is only one source and one destination and so after each completed handshake the wires will be in phase. Generally in processor architectures, data transfers take place over shared highways and the four phases of the protocol map to the four states of bus activity: an inactive state; (either a request is made for data or) data placed on the bus; (data is placed on the bus or) the receiver signals the receipt of data; (the acknowledgement cleared or) data is removed from bus.

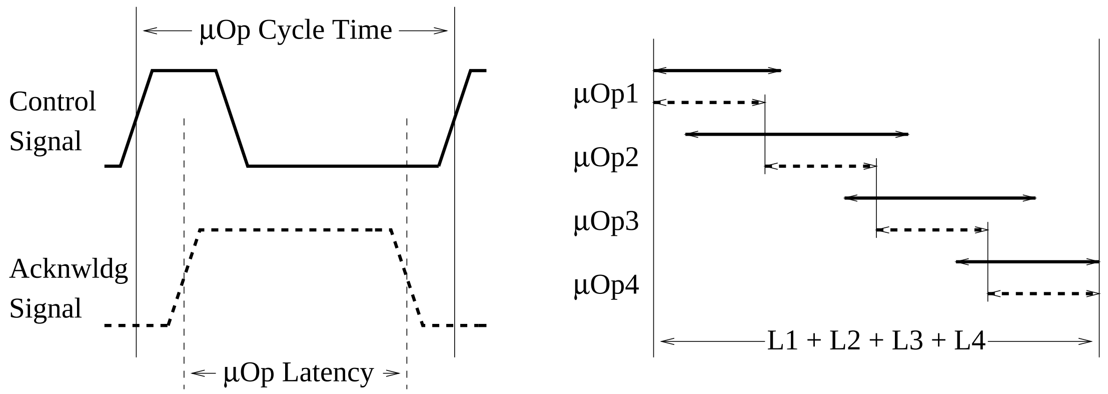


Figure 5–10: Overlapping micro-operation handshake cycles

As suggested earlier, only one of the four phases actually contributes to the progress of the operation. In practice, the overhead of the other phases can be reduced by overlapping them with the micro-operation (computation) associated with the sender's and/or receiver's stage (see Figure 5–10). In a pipeline, overheads can be further minimised by overlapping handshake phases of successive stages. For example, fetching a single operand requires a handshake between the register interface and register bank to access the operand, and the register interface and the functional unit interface to transfer the operand to the appropriate functional unit:

Phase 1 (REGISTER ACCESS) – While the register interface makes a request to accesses a register, the FU interface can initiate the operand fetch handshake by making a request to the register interface.

Phase 2 (DATA TRANSFER) – When the operand is received by the register interface it carries on and completes the handshake with the register bank. Concurrently, if it has received the operand request from a FU interface, then the data is transferred over the bus.

Phase 3 (FM BUSY) – The functional unit interface receives data, it is transferred to its FM and it completes the handshake with the register interface. Meanwhile, the first handshake (register bank-register interface) may have completed, which implies that the next register access could begin.

Phase 4 (FM BUSY and/or REGISTER ACCESS) – When the operand request signal from the FU interface has been cleared, then the register interface removes the data from the bus. Meanwhile, the register interface could also be accessing another register or although the FU may still be busy, its interface could make another request for operands.

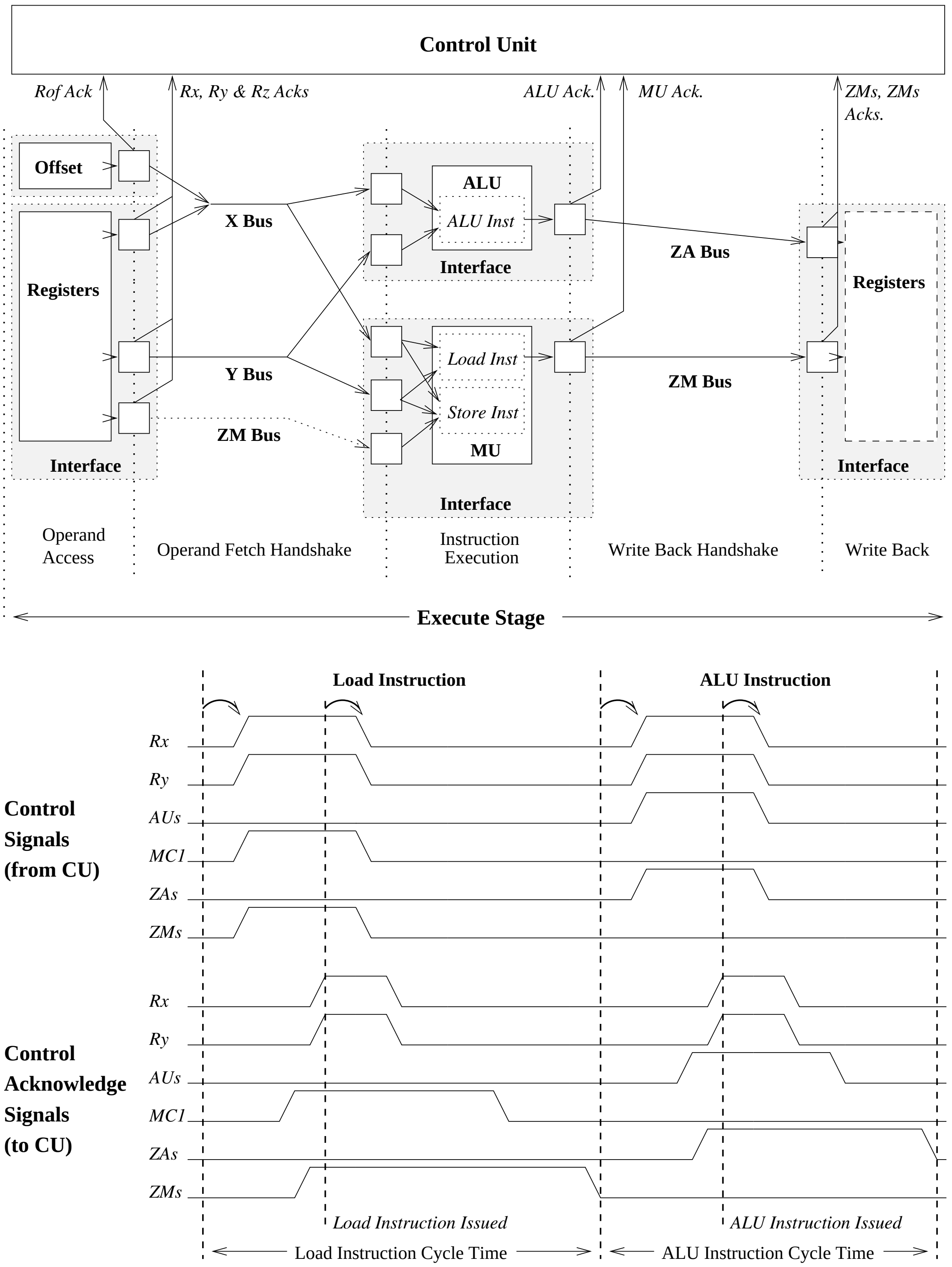
In terms of fetching the operand for the FU, if there were no delays, then this is the shortest time possible (i.e. the sum of the critical latencies). However, the time between operand fetches may be increased by the unnecessary additional time the bus is being driven, this being the transit time of the corresponding de-assertion of the acknowledgement signal and time to remove data from the bus. Notice that in this case, data transfer is request-driven, i.e. for each operand that is required, the FU asserts a request signal over the appropriate bus to the register read port. This ensures that resources (registers and buses) are utilised for no longer than is necessary. The register control signals together with the handshaking protocol prevent bus contention occurring.

5.18.2 Implications for the Compiler

Code scheduling is important in architectures which are able to exploit instruction-level parallelism. In synchronous RISC systems, the order and the time at which each instruction is to be issued are determined by the compiler. Generally, instruction execution is started at the next machine instruction cycle which is determined by the clock. In MAP, instructions are issued in-order as soon as possible – allowing instructions to execute when ready. In the control unit, the effect of this scheduling approach is to initiate the instruction issue immediately after the previous one, thereby reducing the idle time between instructions. For example, given two events: A followed by B, in a synchronous design B will be captured only at the first clock *after* the worst-case delay between A and B. On average, this can still be a significant time after the actual occurrence of B. Thus, in a synchronous design, each instruction will spend a fixed period of time (determined by the largest worst-case stage delay) in each stage regardless of requirement, while in a MAP design, instructions spend varying amounts of time in only the *relevant* stages. Therefore, the asynchronous design is more efficient – each instruction spends only as long as necessary in each stage, and is better able to exploit any concurrency. A side-effect of this is that in MAP, the execution time of instructions cannot be predicted exactly at compile-time. However, a MAP compiler need only generate an appropriate instruction *ordering* to maximise the exploitation of ILP and need not be concerned with the time at which instructions are actually issued.

5.19 Summary

The utilisation of parallelism between instructions in high performance processors is very important. This chapter has investigated the influences of an asynchronous control paradigm on the design and performance of processor architectures for exploiting fine-grained ILP. An ILP MAP design has been outlined and how the control for such an architecture can be implemented efficiently has been shown. The rôle of the CU in an asynchronous processor has been considerably simplified, just to initiating individual micro-operations as early as possible. The control of the datapath is distributed to local interfaces courtesy of the micronet. The advantages of this approach accrue from being able to exploit both the actual run-time delays of the microagents and their concurrent operation. The results show that given some set of architectural resources, an asynchronous control paradigm implemented as a micronet is able to achieve near optimal utilisation efficiently. Furthermore, as one might expect, when a FU operation is the slowest stage in the pipeline, then maximum utilisation can be achieved. The improvement in performance can be accredited to the style of design, which can exploit advances in technology better, unlike current synchronous designs. The next chapter investigates the suitability of MAP architectures as good targets for optimising compilers.

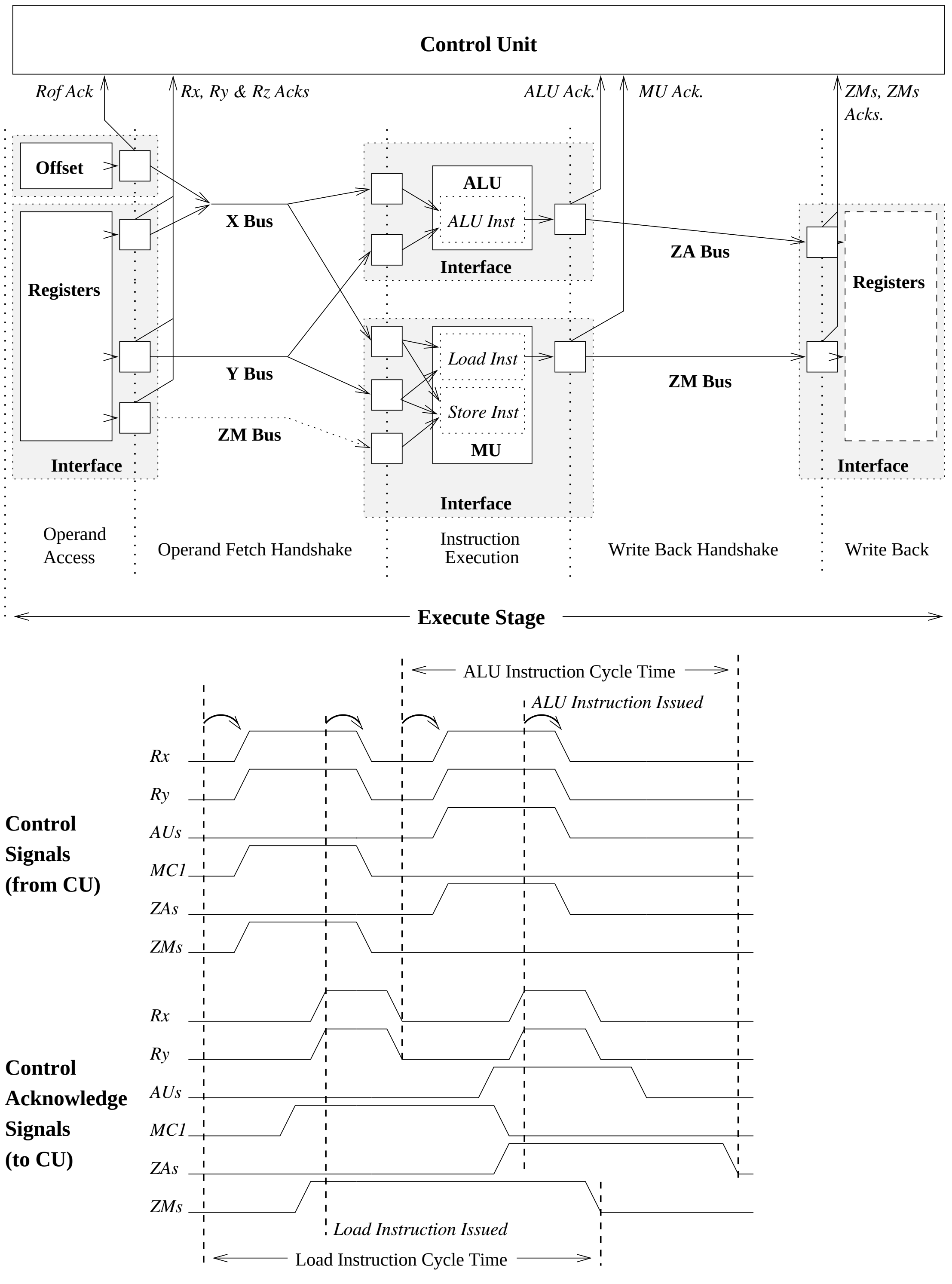


Rx, Ry, Rz Acks. Asserted : Upon access to source register. Deasserted : When Operand Access phase complete.

ALU, MU Acks. Deasserted : When Write Back Handshake complete (LD, ALU) or Instruction Execution finished (ST).

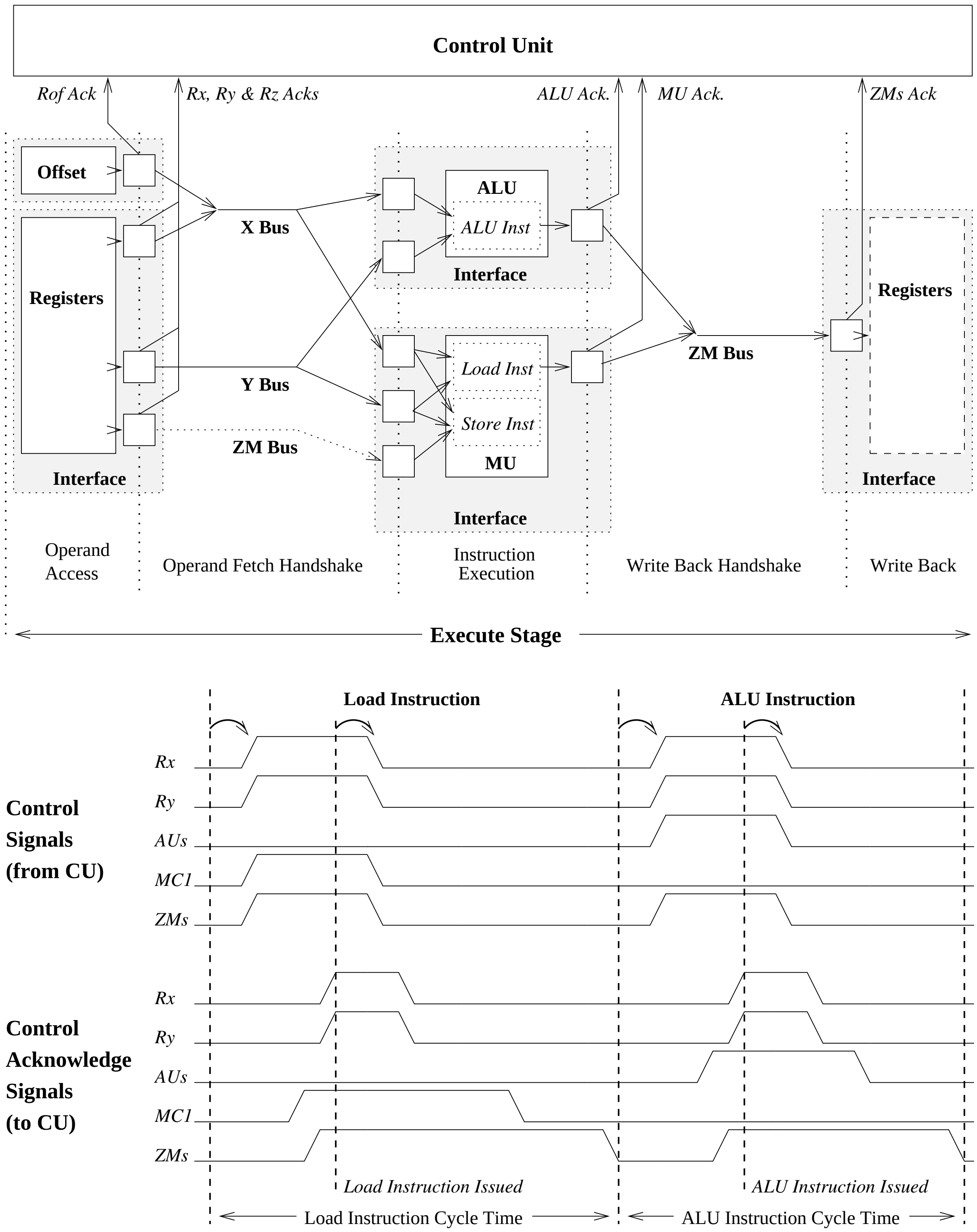
ZMs Ack. Deasserted : When data has been written to its destination register.

Figure 5–11: The micronet model for Refinement Step 1



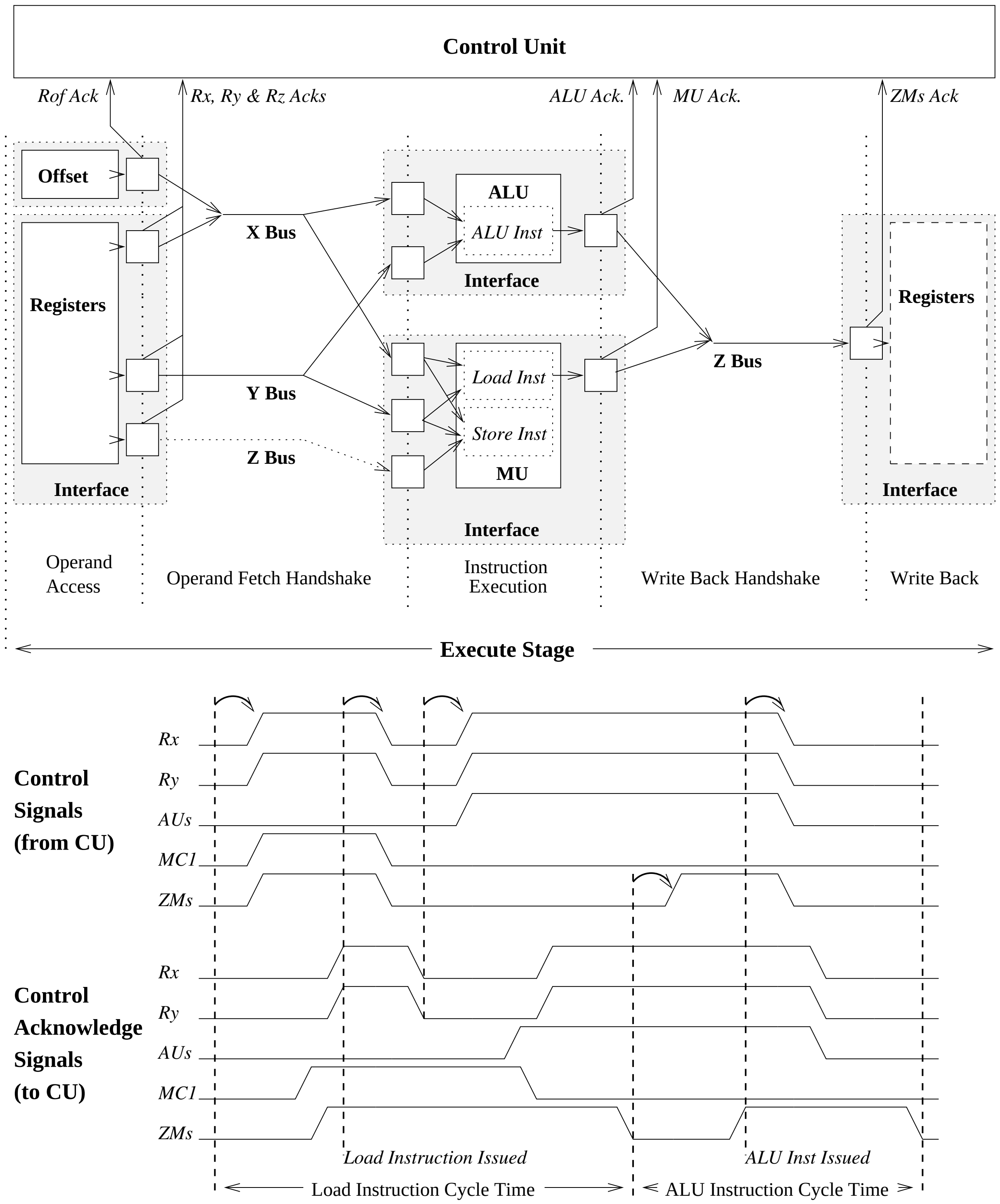
Rx, Ry, Rz Acks.	Asserted : Upon access to source register.	Deasserted : When Operand Access phase complete.
ALU, MU Acks.	Deasserted : When Write Back Handshake complete (LD, ALU) or Instruction Execution finished (ST).	
ZMs Ack.	Deasserted : When data has been written to its destination register.	

Figure 5–12: The micronet model for Refinement Step 2



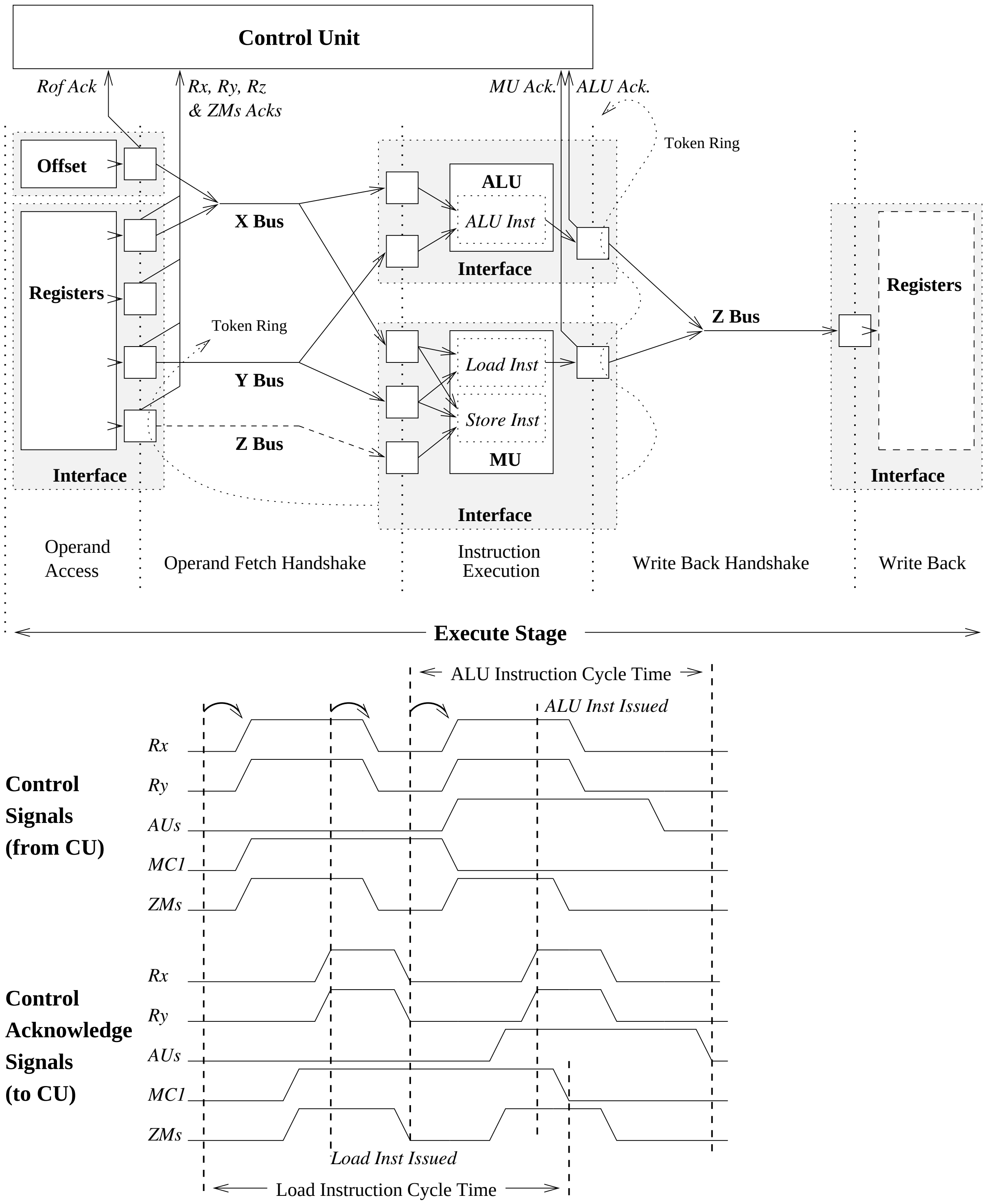
Rx, Ry, Rz Acks.	Asserted : Upon access to source register. Deasserted : When Operand Access phase complete.
ALU, MU Acks.	Deasserted : When Write Back Handshake complete (LD, ALU) or Instruction Execution finished (ST).
ZMs Ack.	Deasserted : When data has been written to its destination register.

Figure 5–13: The micronet model for Refinement Step 3



Rx, Ry, Rz Acks	Asserted : Upon access to the source register. Deasserted : When Operand Access phase complete.
ALU, MU Acks.	Deasserted : When Write Back Handshake complete (LD, ALU) or Instruction Execution finished (ST).
ZMs Ack	Deasserted : When data has been written to its destination register.

Figure 5–14: The micronet model for Refinement Step 4



Rx, Ry, Rz Acks	Asserted : Upon access to the source register. Deasserted : When Operand Access phase complete.
ALU, MU Acks.	Deasserted : When the Functional Unit's execution stage has finished.
ZMs Ack	Deasserted : When the destination register has been locked.

Figure 5–15: The micronet model for Refinement Step 5

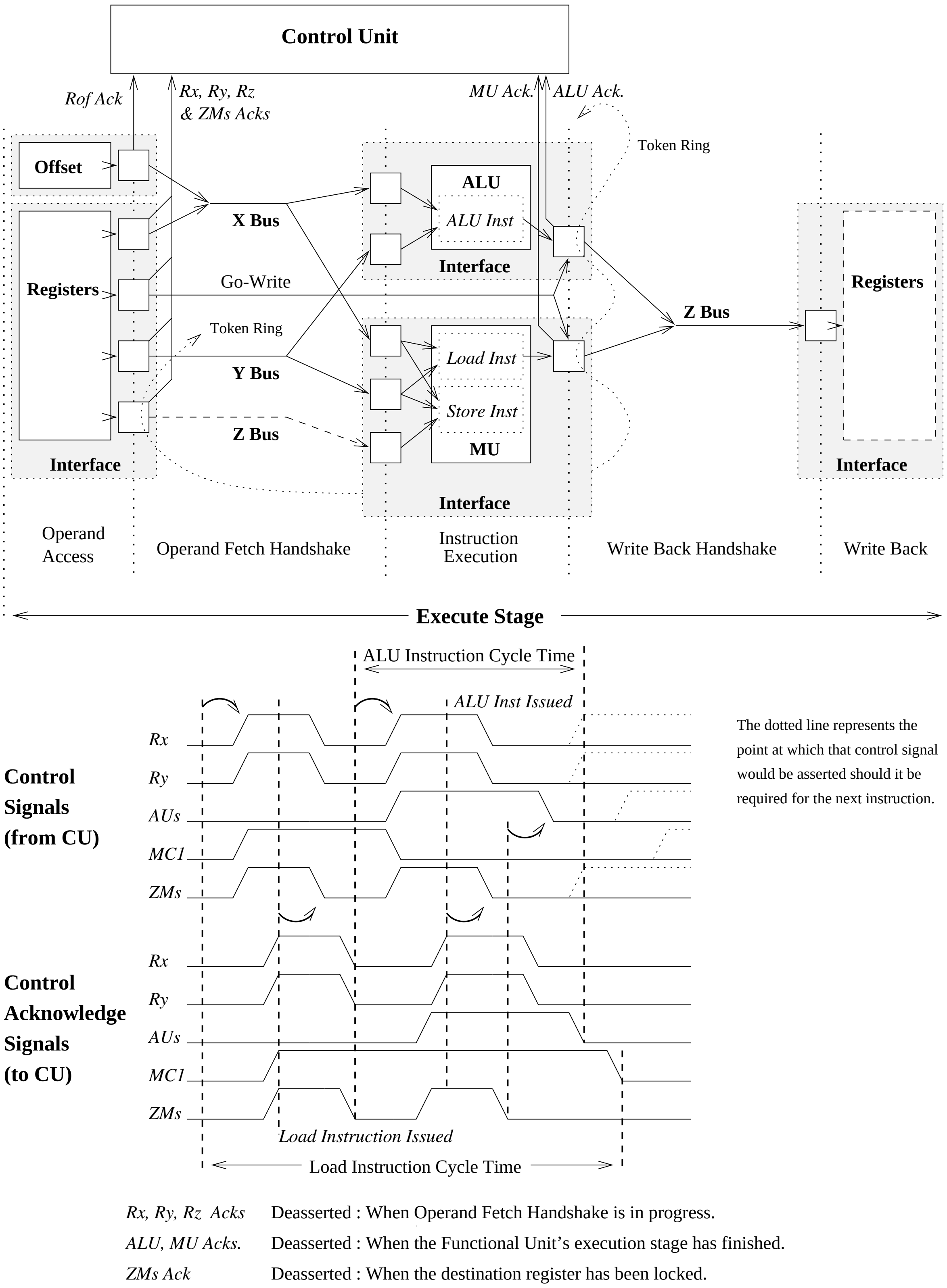
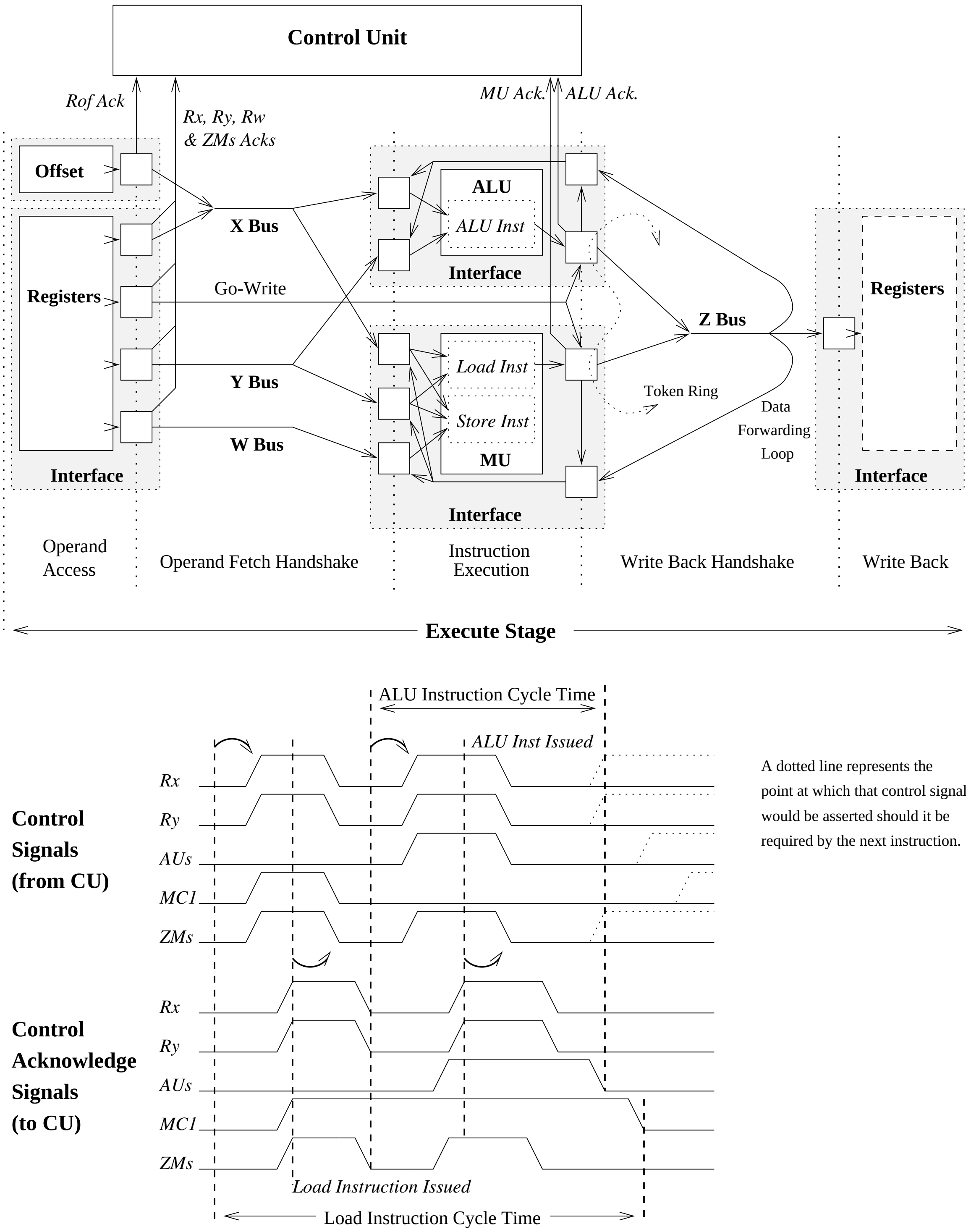


Figure 5–16: The micronet model for Refinement Step 6



Rx, Ry, Rw Acks	Deasserted : When Operand Fetch Handshake is in progress.
ALU, MU Acks.	Deasserted : When the Functional Unit's execution stage has finished.
ZMs Ack	Deasserted : When the destination register has been locked.

Figure 5–17: The micronet model for Refinement Step 7

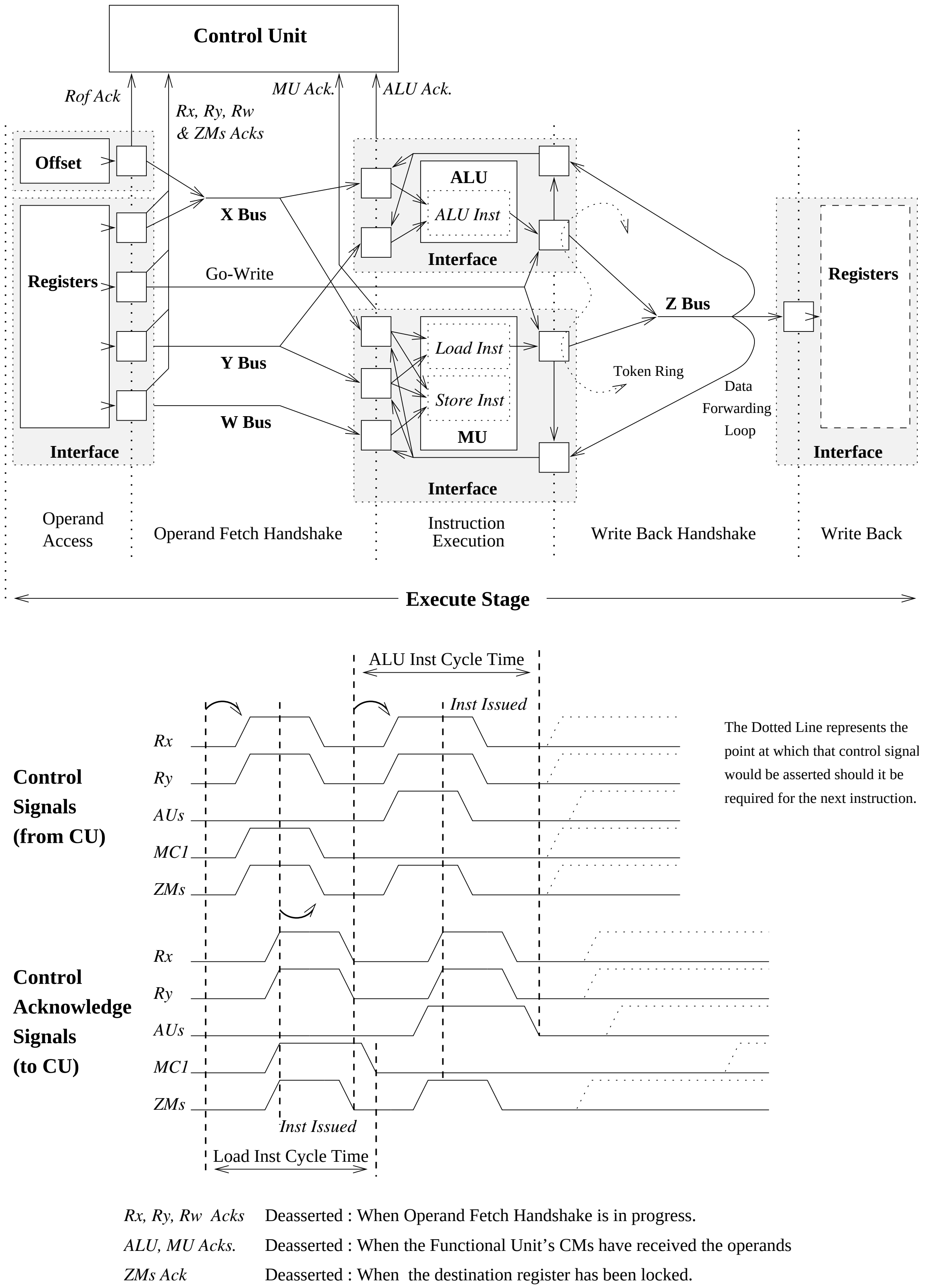


Figure 5–18: The micronet model for Refinement Step 8

Chapter 6

The Control Paradigm and the Compiler

6.1 Introduction

It is important that any processor design be a good target for a compiler, in order that the architectural and technological benefits afforded by the design be efficiently realised. The execution times of programs are strongly influenced by the relationship between the compiler and the rest of the system [176]. In the case of MAP architectures, it is important to understand the influences of asynchronous control on parallelising compilers. The compiler's rôle is to identify parallelism within the program, generate the appropriate code and efficiently schedule the instructions for the given processor architecture. This chapter investigates how these functions are influenced by an asynchronous control paradigm, and examines the design of a static instruction scheduler for MAP architectures.

As described earlier in Chapters 4 and 5, the micronet improves the performance of the instruction set by exploiting average delays and by exposing both spatial and temporal instruction-level parallelism (ILP) within an architecture. This chapter discusses how these advantages can be exploited; a generic computational model for MAP architectures is developed and techniques (heuristics) are introduced which together with the architecture's distributed control strategy allow the compiler to efficiently exploit the available ILP. The intention of this preliminary study is not to propose the best heuristic or scheduling strategy for MAP architectures, but rather to show that a micronet-based datapath can indeed be a suitable target for compilers.

6.2 Compilers

A compiler has three machine-dependent tasks which affect the performance of a program. *Code Generation* determines which instructions implement the given program most efficiently, which can be made simpler by good instruction set design [34]; *Instruction Scheduling* attempts to find an optimal ordering for the chosen instructions; and *Register Allocation* assigns variables to physical registers.

Instruction scheduling is an important feature for processor architectures which exploit instruction-level parallelism, since fast program execution relies on a good code schedule to both reduce the effects of hazards and to maximise functional unit utilisation. In code generation, a particular implementation of a higher-level function is usually determined by the combination of instructions which leads to the minimum execution cost. However in ILP processors, this cost is also affected by the order in which instructions are scheduled.

Instruction scheduling is classified as being *local* if it only considers instructions within a basic block [14] [70], and as being *global* if it considers instructions spanning multiple basic blocks [15] [52]. While local scheduling can extract parallelism within a basic block, global scheduling can exploit further program parallelism by allowing inter-block movement of instructions [3] [46]. Furthermore, in architectures which exploit ILP, in order to generate good code for a particular function, the compiler can no longer just take into account the cost of individual instructions but rather their collective costs as determined by their schedule.

Although register allocation can also introduce hazards due to register dependencies (e.g. anti-dependencies), techniques such as register renaming can be employed to reduce this effect. Note that since global scheduling is generally achieved independently of the architecture, this chapter concentrates on local scheduling which is more machine-dependent. Furthermore, local scheduling is generally used to fine-tune the code produced after global scheduling [70].

6.3 Scheduling Challenges in MAP Architectures

Most modern synchronous processors enhance their performance by exploiting ILP. This is achieved in two parts: firstly, parallelism within the program has to be exposed [9] (e.g. through loop unrolling) and, secondly, a semantically correct instruction ordering has to be achieved which utilises as much of the available parallelism amongst the resources of the target architecture. Although this ordering can be imposed either at compile-time or at run-time, ILP might be best exploited statically rather than dynamically, more so since the dynamic approach cannot exploit a greater degree of parallelism beyond the scope limited by the fetched instructions.

The trend towards static instruction scheduling, i.e. the reliance on the compiler to generate the optimal schedule, has been aided by the predictability of execution costs on synchronous processors. The optimising compilers for synchronous pipelines assume a deterministic behavioural model of the target with each stage delay being approximated to being the same, having been fixed *a priori* by the clock. In contrast, a linear, asynchronous pipeline, e.g. *micropipeline* [158], has stages whose delays can vary. A compiler in this case has a less accurate timing model of the target, and any optimisations based on a synchronous model, such as scheduling instructions in execution gaps as found in the MIPS re-organiser [13] [71], are less effective.

A micronet enables the exploitation of both spatial and temporal concurrency between instructions (in contrast, a micropipeline only exploits temporal parallelism). Therefore, it is less easy for a compiler to predict the behaviour of the micronet for the following reasons: firstly, as in a micropipeline, the delay of each pipeline stage might vary; secondly and more uniquely, each instruction only visits the relevant stages and the multiple paths enable more than one instruction to operate concurrently within a stage, which enables instructions to race each other, with possible out-of-order completion of instructions. Furthermore, instructions may interfere with each other when competing for the same resource in a particular stage.

The effective performance which a MAP system can deliver depends intimately on the compiler's ability to match the parallelism in programs with the temporal and spatial concurrency exposed by the MAP architecture. The resulting instruction schedule should aim to keep the functional units busy thereby increasing their utilisation and improving the overall performance. Unlike synchronous schedules which imply both an order of execution for the instructions and the times in terms of multiples of the basic instruction cycle, when they are to execute, asynchronous ones only imply an order and are efficiently issued "dynamically" by the control unit (CU). This removes the need for the inclusion of NO-OP instructions in asynchronous schedules. Note that in synchronous designs, the selection of which instruction to issue in a given cycle is generally performed at compile time in superpipelined (and VLIW) machines and at run-time in superscalar ones.

6.3.1 MAP Behaviour

A MAP architecture has several communicating pipelines all of whose stages can potentially be busy simultaneously. The task of the scheduler is to order the instructions in such a manner so as to maximise the resource utilisation, minimise the resource contention and allow the processor's control unit to maintain an optimal instruction issue rate. The control unit issues successive instructions as early as possible in order to initiate the instruction's execution immediately after the previous issue, thereby reducing the idle time between instructions.

A micronet can be stalled due to contention for resources. In particular, the CU (also referred to as the *issue unit*) will be stalled when the resources required by the current instruction are all busy. The scheduler attempts to minimise this by suitably ordering the instructions at compile-time. If it is impossible to schedule successive unrelated instructions, then the micronet minimises the stall at run-time. In the case of data-dependent instructions: both instructions are issued, with the second instruction awaiting the result to be forwarded. In the case of resource contention: the second instruction performs all the micro-operations up to the microagent which is busy. In effect, only the offending micro-operation is stalled, rather than the entire instruction. These fine-grained hazard avoidances are enforced at run-time by the pre-issue conditions of the micronet as previously described in Chapter 5.

6.3.2 A Parameterised Computational Model

A computational model describes the scheduler's view of the target architecture. The model is the basis upon which the scheduler aims to maximise the amount of parallelism that can be exploited. One of the advantages afforded by asynchronous and distributed control in the design of processors is the modularity and composibility which allows designers to easily modify and explore the architectural design space e.g. determining the optimal number of resources for a class of problems. It would be advantageous, therefore, not to have to redesign the scheduler each time as well. This will need two requirements to be fulfilled: the scheduling strategy should not be specific to any particular architecture; and the computational model should capture the salient characteristics of any target architecture (the *holy grail* in the field of scheduling [134]).

For synchronous architectures the computational model is simple: instructions do not interact and their execution times are considered fixed. In contrast, the model for a micronet-based processor is necessarily less accurate for the following reasons: execution times for even the same instruction may vary due to data-dependent operations, environmental parameters, and the interactions between different instructions which are executing simultaneously. However, the modularity and composibility of the micronet makes it easy to parameterise the computational model which would allow the same scheduler to be applicable to a variety of architectures. Unfortunately, this concept cannot be easily adopted for synchronous architectures since each one is almost unique because of its centralised controls and any changes made can effect the behaviour of the whole design.

The MAP computational model views an architecture as a collection of resources or microagents (a number of issue units, a number of various functional units, and a number of bus highways) which are connected in some fashion (i.e. some functional units may share buses, while others have dedicated point-to-point connections). Each microagent associates a latency and a cycle time with each of its micro-operations, the former determines when the result becomes available and the latter is the rate at which those micro-operations can be processed. The model currently assumes a "five-stage" network (register access, operand fetch handshake, execution, write-back handshake, and write-back) (see Chapter 5) to which resources are allocated depending on their type – issue unit; register bank; operand fetch bus; functional unit; or write-back bus. The register file can be modelled as one large file or a number of smaller ones depending on the number of operand fetch ports. It is also possible to model VLIW or superscalar architectures. The parameterised model effectively forms a resource graph of the target architecture. In general, this graph is irregular and does not have the same conventional connectivity patterns that are normally associated with multiprocessor scheduling graphs, e.g. full connection, grid, hypercube or a ring topology.

6.4 The Scheduling Problem

The MAP scheduling problem can be stated as follows: *Given a set of heterogeneous resources with variable execution times, devise a minimal-length, non-preemptive schedule which respects dependencies within programs; each program being described as an arbitrary partial ordering of instructions.*

This type of problem, usually referred to as the precedence- and resource-constrained instruction scheduling problem has been studied well, and it is known that even by imposing restrictions, the problem is still NP-hard [32] [85] [168]. For example, when the execution times of tasks are not uniform and their partial order is arbitrary, then for two or more identical processing units, the problem of determining a minimal-length, non-preemptive schedule is NP-complete [59]. This result is true even if all of the tasks are independent. Therefore, in order to achieve near-optimal execution times for given applications on MAP architectures, an efficient (polynomial-time) scheduling algorithm based on one or a number of heuristics must be devised.

6.4.1 Similar Scheduling Problems

The scheduling of instructions for MAP finds echos in other scheduling problems:

1. Multiprocessor Scheduling: There is a wealth of strategies and solutions to various classes of scheduling problems. For example, multiprocessor scheduling considers tasks as the basic unit of work, whether one considers processes, code segments, or even machine-code instructions they can all be viewed as tasks at a different levels of granularity. These problems usually assume that processors are homogeneous (i.e. identical), whereas a MAP architecture has different functional units each of which can only execute a unique set of instructions. Furthermore, since multiprocessor scheduling only considers acyclic dependencies between tasks and that each task is only executed once, this technique can only be used to schedule instructions within basic blocks.

    - Level Scheduling was an early approach used in operational research and assembly line problems [75]. This scheme is only optimal when considering unit execution time (UET) systems and tasks graphs which are either in- or out-forests. Priorities are assigned to all tasks: the tasks within the same level of a directed acyclic graph (DAG) being assigned the same values and the higher levels within the DAG (those farthest from the terminal level or sink tasks) being given higher priorities. The highest, unexecuted, ready task, i.e. a task which has no predecessors or all of its predecessors have already been executed, is assigned to the first processor which becomes available. More recently, optimal solutions for arbitrary-shaped DAGs for up to 2 processors have been found [33] [57] [151].

2. Graph Colouring is a technique used in register allocation [27]. A large number of symbolic registers are mapped onto a limited number of physical registers in a CPU. At any time t there are a number of "live" symbolic registers which need to be optimally allocated. Similarly in MAP, at any time t there is a list of instructions that are eligible to be issued for execution. The choice of instruction for scheduling depends on availability of resources and the cost, of say, not scheduling the instruction immediately.

3. In dataflow machines, instructions are issued as soon as their operands are available. This is achieved completely dynamically in hardware but incurs significant run-time (book-keeping) costs. Scheduling in traditional synchronous RISC architectures is achieved completely statically. An effort to reduce the book-keeping costs has lead to an interest in dataflow-RISC hybrids [58]. MAP architectures can also be viewed as a hybrid of these two classical styles. As in the RISC architectures, code scheduling is done statically but, additionally, instruction issue (and even possibly the instruction schedule) is fine-tuned dynamically to take advantage of run-time characteristics as in the data-flow model. Notice that in some sense, MAP is more interested in dataflow at the microagent-level than at the instruction-level. This now begs the following question: How much scheduling **should** be done statically in the MAP scheduler and what should be left for the MAP hardware? Before this can be answered, the rest of this chapter attempts to determine how much scheduling **can** be done at compile-time.

6.5 A Scheduling Methodology for MAP

A directed acyclic graph (DAG) is used to represent the instructions within the basic blocks of a program. (Techniques such as trace scheduling [46] [52] or global compaction [130] could be used to increase the size of these blocks.) Each node within the DAG corresponds to an instruction, and each edge to a data dependence between instructions. Typically, an instruction cannot begin execution until all of its predecessors have completed and their results have become available. In practice, it is not necessary to stall the instruction completely in all the cases where such dependencies exist. Since the micronet already minimises the length of any stall, i.e. only stalling until their dependencies have been resolved, the implicit (and possibly) unnecessary stalls incurred by a conventional computational model, which may adversely affect the optimality of the schedules generated by heuristics, can be avoided. The implications for the MAP scheduler, i.e. the degree to which an instruction needs to be stalled, depends on the type of dependency implied by the edge within the DAG, as described as follows:

Read-after-Write – Although the dependent instruction will be issued, its execution will be delayed (by the micronet) until the completion of its predecessor. In practice, it is preferable not to issue such an instruction, since some of the resources earmarked for the dependent instruction will become unavailable for use by other, now "ready-to-execute" instructions, which might introduce further structural hazards in the bargain.

Write-after-Write – Only the write-back order has to be maintained and this is also achieved in hardware by the micronet. Two instructions are permitted to execute concurrently. Although all of the second instruction's micro-operations will have been initiated, the write-back micro-operation will stall for as long as the first instruction holds on to the destination register. The current MAP architecture supports only one outstanding register lock request, therefore a subsequent third instruction which requires a locked register cannot be issued, until the first write-back has been completed. The scheduler should avoid arranging instructions which write to the register file immediately after two instructions with write-after-write dependencies if independent instructions cannot be found for issue between the two dependent instructions.

Write-after-Read – In the case of an architecture with a single set of operand fetch buses, the hardware ensures that a dependent instruction will be unable to lock its destination register before its predecessor has fetched its operand. Should there be a number of operand fetch buses (as in a superscalar MAP), and the possibility of a dependent instruction obtaining its operands before its predecessor, then this instruction may have to be stalled. This would only be necessary when the time to execute the dependent instruction is less than the operand fetch time for the predecessor. This hazard is also known as an anti-dependency, and along with write-after-write hazards can be avoided by register renaming.

Hazard resolution is a good example of the interaction between the compiler and the architecture. Since there is no concept of time in the schedule, it is impossible to avoid all hazards at compile time (c.f. the MIPS organiser). The scheduler can only hope to produce an ordering of instructions which reduces the number of hazards, and relies on the MAP architecture to minimise their effects by efficiently resolving them in hardware.

In MAP architectures, it is better to schedule independent instructions successively since this may allow the optimal instruction issue rate to be achieved. In practice, finding independent instructions is not always possible. With the MAP scheduling problem being NP-complete [59], heuristics are required to map tasks from a program DAG on to a resource graph. The method which has been investigated here, combines some elements of the approaches described earlier but is based primarily on the well-known List Scheduling method.

6.5.1 The Scheduler

List scheduling (LS) is a general method for scheduling tasks in resource-constrained problems [32]. LS builds a *ready set* that contains all of the tasks which are not waiting on the results of other tasks. When a processor becomes available, a task with the highest priority is chosen from the set and assigned to it. The ready set is obtained from a topological sort of the data dependence graph. LS relies on other heuristics to prioritise the ready tasks and guide it towards an optimal solution. This has lead to a profusion of LS-based heuristics [12,45,77,104,134].

The MAP solution adopted here is based on a greedy scheduling algorithm for list scheduling which was proposed by Coffman and Graham [33]. This is an optimal, $O(n^2)$ algorithm for arbitrary precedence constraints on two processors with unit execution costs. A MAP scheduler has to deal with heterogeneous resources and can no longer just choose the ready instruction with the highest priority, but must also consider whether the correct resources are also available i.e. the instruction must be *executable*. Once an executable instruction is issued, its execution cannot be suspended and resumed at the point of suspension at a later time, i.e. schedules must be non-preemptive. The goodness of these schedules are highly dependent on the parameter(s) that are used to prioritise instructions within the ready list [1] [112], and these MAP-specific heuristics are discussed in the following sections.

Compared to multiprocessor environments, although the scheduler for MAP does not have to explicitly consider interprocessor communications it does however effectively assume data is not local since operands have to be fetched from and sometimes returned to the register bank (i.e. incurring some cost). Note also that even though data forwarding might be considered to be equivalent to local data access, it is not modelled in the computational model since this is an architecture-specific feature (i.e. not permitted in the parameterised model) which is impossible to predict *a priori*.

Minimising Idle Times

The scheduler's first assumption is that minimising the stall time will lead to better (or at least near-optimal) program execution time (the first priority heuristic). This implies that the MAP compiler should not schedule instructions until their dependencies have been resolved (as discussed in Chapter 5 and Section 6.5) and the necessary microagents (resources) are available. This requirement is met by basing the heuristic's cost function on worst-case instruction execution times (see Section 6.7.1 for further details). This implies that the computational model has to maintain a scoreboard of resource activities.

Primary Instruction Priority

In Coffman and Graham's algorithm, interprocessor communication is assumed to be zero and tasks have unit execution times, which means that time can be conveniently treated as being discrete rather than continuous. This allows priorities to be assigned based on the task's level within the DAG from the sink tasks. Since instructions have different worst-case execution times in MAP, the problem is similar to multiprocessor scheduling with interprocessor communication delays (where communication costs are only incurred if dependent tasks are scheduled on different processors). The solutions adopted in this field have been based on critical path analysis and heuristics [62] [91] [148]. (The critical path cost of a task is the largest sum of costs along a path from itself to a sink task.) In the MAP computational model, although actual instruction execution costs may vary, these critical path costs can be determined *a priori* by basing them on fixed, worst-case instruction costs.

Secondary Instruction Priorities

The heuristics applied so far may still not prioritise the executable tasks sufficiently. Therefore, additional heuristics are required to further prioritise the candidate tasks. One feature which does seem to significantly influence the best choice of candidate is the dependents of the chosen task. The two heuristics used to "break ties" amongst candidates of the same priority act as follows: the first one gives a higher priority to the task with the larger number of successors which are *solely* dependent on it. A feature of this heuristic is that the priority of a task increases with time. If a tie is still unbroken, then a higher priority is given to the task with the most successors. Additionally, these heuristics highlight the need to consider not only which tasks need to execute in the future, but also their resources.

The Importance of the Instruction Issue Cycle Time

Unlike synchronous pipelines, micronet resources have two parameters which affect instruction execution costs: the micro-operation's latency and its cycle time. Together with program parallelism and the number of resources, a limiting factor on the amount of exploitable ILP is the cycle time of the issue unit in relationship to the execution time of instructions (or more accurately their cycle times).

In order to minimise the issue unit's stall time, the compiler has to devise a schedule that allows instructions to be issued continuously at the highest possible rate, which is equivalent to one every minimum Instruction Issue Cycle Time (IICT). Synchronous datapaths are pipelined or where necessary super-pipelined (i.e. the functional units are themselves pipelined) sufficiently to achieve this goal. Due to the spatial ILP in MAP, instructions are issued at a rate (determined by the IICT and dependencies) which is faster than their Instruction Cycle Times (ICTs). The ICT is the effective issue time (due to pipelining) for a particular instruction, which is determined by the rate at which that specific instruction type can be processed. As the IICT, which is less than the largest ICT, gets smaller, the MAP architecture behaves more in a superscalar fashion and therefore the value of the IICT itself can have a significant influence on the optimality of a schedule. This is less significant when the IICT is comparable to the largest ICT, in which case the order of the independent instructions is less critical, since the micronet behaves like a linear pipeline without any spatial concurrency.

IICT, ICT and Lookahead

When choosing an instruction to schedule, it may be beneficial to consider not only those instructions which are ready, but also the ones which will become ready in the near future, called *instruction lookahead*, e.g. within the next minimum IICT. Note that this may mean deliberately selecting an instruction that causes the processor's issue unit to stall.

The two steps of choosing an instruction and checking to see if sufficient resources are available for it should not take place independently. Since the scheduling of an instruction is subject to current resource availability, the scheduler should also consider future resource requirements (Resource Lookahead). Example 1 and Example 2 contrast the influence of IICT and resource lookahead on determining an optimal schedule. A_1 and B are ready candidate instructions, with a third instruction, A_2, which has a structural dependency on A_1.

Example 1 : ***Resource Lookahead***

switch IICT
  case 0: *Choose schedule* $\{A_1,B,A_2\}$ *or* $\{B,A_1,A_2\}$;
    * *Either schedule is optimal* *\
  case ($0 \leq$ IICT $< \frac{1}{2}\cdot$ICT$_A$):
    if (ICT$_B$ $>2\cdot$ICT$_A$) *Choose schedule* $\{B,A_1,A_2\}$;
      * *Instruction B takes longer than the both* A_1 *and* A_2 *\
    else *Choose schedule* $\{A_1,B,A_2\}$;
    * *In other words, combine the resource requirements of* *\
    * *dependent instructions and schedule the instruction* *\
    * *according to the resource with the most work.* *\
  case ($\frac{1}{2}\cdot$ICT$_A$ $\leq$ IICT $<$ ICT$_A$):
    if (ICT$_B$ $>2\cdot$ICT$_A$) * *then schedule* B *first (as before)* *\
      Choose schedule $\{B,A_1,A_2\}$;
    else * *schedule* A_1 *first* *\
      if (ICT$_B$ $<$ ICT$_A$) *Choose schedule* $\{A_1,A_2,B\}$;
      else *Choose schedule* $\{A_1,B,A_2\}$;
  case (ICT$_A$ $\leq$ IICT):
    * *Schedule the instruction with the largest* ICT *first* *\
    if ICT$_A$ $<$ ICT$_B$ *Choose schedule* $\{B,A_1,A_2\}$;
    else *Choose schedule* $\{A_1,A_2,B\}$;
end switch;

In the case of scheduling heuristics which do not consider resource lookahead, the schedules they generate might be as follows:

Example 2 : ***Without Resource Lookahead***

if (IICT = 0) *Choose schedule* $\{A_1,B,A_2\}$ *or* $\{B,A_1,A_2\}$;
  * *Again, either schedule is optimal* *\
else * *Simply schedule the instruction with the largest* ICT *first.* *\
  if (ICT$_A$ $<$ ICT$_B$) *Choose schedule* $\{B,A_1,A_2\}$;
  else if (IICT $<$ ICT$_A$) *Choose Schedule*$\{A_1,B,A_2\}$;
  else *Choose schedule* $\{A_1,A_2,B\}$;

The lookahead heuristics attempt to match the available program and architectural parallelism over a short window of time. The strategy of repeating the process over the entire program allows the instruction-level parallelism to be exploited more evenly. This has two effects: firstly, a better program makespan is usually achieved; and secondly, a schedule is generated which is more robust to deviations from the predicted instruction costs because only the appropriate amount of program parallelism is exposed which can be exploited by the target at any one time. Since costs are based on worse-case values rather than typical ones, the traditional list scheduling heuristics tend to overly migrate independent instructions to the top of the schedule, leaving insufficient parallelism for later. Kerns and Eggers [88] proposed a code scheduling algorithm called *balanced scheduling* for synchronous architectures which is similar in concept. Their algorithm is specifically designed to tolerate a wide range of variance in load latency, e.g. cache misses/hits, global and local memory. In these architectures, instruction costs are well defined and considered fixed. Usually the latencies reflect the most optimistic execution, e.g., the time of a cache hit rather than a cache miss. Traditional schedulers improve performance through reordering instructions to avoid pipeline stalls, e.g., by inserting independent instructions after loads to keep the CPU busy. The number of instructions inserted (in the best case) depends on this latency value. If the load instruction is delayed beyond the scheduler's estimate, then the processor will stall. However, if the latency is shorter, the destination register of the load instruction will be tied up for longer and this may increase register pressure enough to cause unnecessary code spills. Unfortunately both balanced scheduling and resource lookahead are computationally more expensive than the traditional list scheduling approach and will not be considered further in this initial study.

The approximation algorithm

The algorithm takes as its input a directed graph of instruction dependencies and a resource graph with architectural parameters, and generates an instruction schedule for the given MAP architecture. Two lists are defined as follows: the WI_list – the list of instructions still awaiting their operands, and the EI_list – an ordered list of instructions which are ready, or will be ready in the near future (for lookahead instructions), but still awaiting issue. The order of the latter list is determined by the critical path costs of instructions, i.e. the primary priority. Next, a prioritised list of executable instructions is derived from the EI_list based on the availability of their resources at the current time. If there are ties, an instruction (or instructions in the case of superscalar MAP) is chosen for issue based on secondary priority values.

The scheduler mimics the behaviour of the architecture's issue unit. The function *generate_schedule()*, as shown in Algorithm 2, schedules instructions based on their readiness, their priority and the availability of resources. Unlike schedulers for synchronous machines, the scheduling of instructions does not proceed in uniform time steps, but rather in an asynchronous event-driven manner until all the instructions are scheduled. Each iteration of the main loop (the `while do` loop in line 5) corresponds to an instant in time when the issue unit is ready to issue an instruction. However, a situation may arise when at some given time there are no instructions ready for issue (line 8), in which case the clock must be advanced, but only as far as necessary to remedy this. The incrementing of the clock simulates the issue unit being stalled. The routine, *advance_clock()*, finds the earliest occurrence of three types of events: the ready time of an instruction in the WI_list and of a lookahead instruction in the EI_list; the time when the result of an operation becomes available in the register file; and the time a busy resource becomes free. Only the first two events can change the status of the EI_list. There is a choice of heuristics which can be applied, either the instruction lookahead or the traditional priority-based approach. Instruction lookahead (lines 9 – 17) chooses the best instruction to issue from the EI_list based on the lookahead heuristic. The function, *get_ready_instr()* described in Algorithm 3, returns from the given list of instructions the one with the highest estimated-time-to-completion (ETC) priority for which there will be sufficient resources in the datapath when issued at its earliest issue time. In the current implementation of the lookahead heuristic, only one instruction is chosen per issue cycle iteration. The routine, *apply.lookahead()* as described in Algorithm 4, implements the instruction lookahead heuristic. The alternative heuristic (lines 18 – 29) chooses the instruction with the highest priority which can be issued immediately. This may involve choosing one or more from a number of instructions with the same primary priority value (ETC). Line 19 creates a list of ready instructions with the same, highest ETC values and line 22 removes those instructions with insufficient resources for issue at the current time. Line 23 supports architectures which incorporate lockstep superscalar instruction issue. The routine *issue_all()* issues as many of the instructions as possible from the given list. If there are not enough issue-slots for the complete list (*rdyI*), then the routine *choosing_insts()* returns the best instruction for issue based on the secondary priorities. The two loops (lines 26 and 27) repeat until either the issues slots are filled or their respective lists become empty. The clock is advanced appropriately depending on whether or not the scheduler was able to issue one or more instructions at the current time (lines 28 and 29). The routine, *update_writeback*, models the behaviour of the portion of the micronet not directly controlled by the issue unit, e.g. write-back bus. Line 32 updates the instruction lists and the next instruction issue cycle iteration begins at a new time.

Algorithm 2 : ***The MAP scheduler (generate_schedule())***

```
curr_time := 0;
calc_completion_times(); \* Critical path analysis for each instruction *\
update_WI(WI_list); \* Determine instruction start times *\
update_EI(WI_list); \* Move ready instructions to EI_list *\
while (WI_list ≠ {}) or (EI_list ≠ {}) do
   no_issued := 0; \* Number of inst issued simultaneously at this time *\
   candidates := EI_list;
   if (EI_list = {}) curr_time := advance_clock(YES, YES, NO, curr_time);
   else if (lookahead = YES) \* Use Instruction Lookahead Heuristics *\
      BestChoice := get_ready_instr(candidates); \* Inst with the highest *\
      \* priority in the candidates list for which there are sufficient resources *\
      if (BestChoice ≠ NULL)
         while candidates ≠ {} do
            NextInst := get_ready_instr(candidates);
            if (NextInst ≠ NULL)
               BestChoice = apply.lookahead(BestChoice, NextInst);
            end while
            if (BestChoice.rdy_time ≤ curr_time + issue_cost)
               issue_instruction(BestChoice); no_issued++;
               EI_list := EI_list - {BestChoice};

      else
         do \* Alternative strategy without Instruction Lookahead *\
            \* same_ETC_list is the list of the highest ETC cost, ready insts *\
            ∃ same_ETC_list ⊆ candidates, s.t. ∀ i ∈ candidates,
               ∃ v ∈ same_ETC_list, s.t. (v.ETC ≥ i.ETC);
            candidates := candidates - same_ETC_list;
            do \* Remove instructions without sufficient resources *\
               ∃ rdyI ⊆ same_ETC_list, s.t. ∀i ∈ rdyI,
                  find_avail_FU_resources(i, datapath, curr_time);
                  if (|rdyI| ≤ spsclr_deg - no_issued) issue_all(rdyI, no_issued);
                  else \* choose between insts in rdyI list *\
                     inst_chosen := choosing_insts(rdyI, no_issued);
                     EI_list := EI_list - {inst_chosen};

            while ((no_issued < spsclr_deg) and (same_ETC_list ≠ {}));
         while ((no_issued < spsclr_deg) and (candidates ≠ {}));
         if (no_issued > 0) curr_time += inst_issue_cycle;
         else curr_time := advance_clock(YES, YES, YES, curr_time);
```

```
            end if
            update_writeback(datapath);
            if (WI_list ≠ {})
                update_WI(WI_list); update_EI(WI_list);
        end while
        update_writeback(datapath);
```

The function described in Algorithm 3 returns, from the given list, the instruction with the highest estimated-time-to-completion (ETC) priority for which there will be sufficient resources in the datapath if it is issued at its earliest issue time. If this time is not the same as the current issue time (i.e. the next earliest scheduling time for any unscheduled instruction), then issuing this instruction will effectively cause the issue unit to stall. However, in practice it is not possible to predict what will actually transpire unless the actual delays can be determined *a priori*. Notice that the scheduler must take into account the fact that some instructions will begin to be issued before they are ready or all of their resources are available which effectively allows the cost of issuing the instruction to be hidden.

Algorithm 3 : ***The MAP scheduler (get_ready_instr(inst_list))***

```
∃ inst ∈ inst_list s.t. inst.ETC is maximum;
\* i.e. inst is the first instruction in the ordered list inst_list *\
while ((inst_list ≠ {}) and (not cand_found))
    if (inst.rdy_time > curr_time + issue_cost)
        \* This instruction can be issued early to hide the issuing cost *\
        inst.issue_time := inst.rdy_time - issue_cost;
    else inst.issue_time := curr_time;
    if (find_avail_resources(inst, inst.issue_time) ≠ {}) cand_found = TRUE;
    else
        inst_list := inst_list - {inst};
        ∃ inst ∈ inst_list s.t. inst.ETC is maximum;
    end if
end while
return(inst);
```

In certain cases it is more prudent to stall the issue unit until a higher priority instruction becomes ready, rather than immediately issuing another ready instruction. The routine, *apply.lookahead()*, as described in Algorithm 4 implements the instruction lookahead heuristic which uses the ETC priority and the earliest issue time of two instructions to determine which of them should be issued first. By comparing the estimated execution time of the two instruction schedules, the order with the smallest time is chosen. Should the two schedules have the same time, then the order where an instruction completes earlier is chosen, since this would at least allow its dependents to become ready sooner. However, if a tie still exists then the secondary instruction priorities are applied to choose a candidate.

Algorithm 4 : ***The MAP scheduler (apply_lookahead(instA, instB))***

```
opt1 := instA.issue_time + instA.ETC;
opt2 := instA.issue_time + instB.ETC + instA.issue_cycle;
opt3 := instB.issue_time + instB.ETC;
opt4 := instB.issue_time + instA.ETC + instB.issue_cycle;
etc_ABl := max(opt1, opt2);
etc_ABs := min(opt1, opt2);
etc_BAl := max(opt3, opt4);
etc_BAs := min(opt3, opt4);
if (etc_ABl < etc_BAl) return(instA);
else if (etc_ABl > etc_BAl) return(instB);
else if (etc_ABs < etc_BAs) return(instA);
else (etc_ABs > etc_BAs) return(instB);
else return(break_ties(instA, instB));
```

6.6 Results

In this section, the makespans of MAP schedules for a number of typical instruction DAGs (briefly described below) are compared with their optimum. The optimal makespan of each DAG is derived from an exhaustive search of all possible valid schedules. The DAGs represent a selection of graph shapes typical of program applications:

BT3 – A Binary Tree with three levels.

BT3.5 – A Binary Tree with three and half levels.

BT4 – A Binary Tree with four levels.

DD – Diamond DAGs which are commonly found in the evaluation of partial differential equations.

DM – Dense matrix multiplication.

SM – Sparse matrix multiplication.

CC – Mix of Load, Store and ALU instructions with data dependencies. (The Hennessy Test used in Chapter 5.)

CCL – A loop unrolled version of CC (i.e. two iterations of the Hennessy Test).

Min1 – This architecture contains the minimum resources – one ALU and one Memory Unit (MU) which both share a single write-back bus. The cycle times and latencies of the ALU, the MU and the write-back micro-operations are assumed to the same.

3bus1 – This architecture has an additional ALU and each of the three functional units has a dedicated write-back bus. (The micro-operation cycle times and latencies are the same as Min1).

Min2 – Same as Min1, except that the micro-operation costs of all of the microagents reflect realistic costs obtained from SPICE-level simulations.

3bus2 – Same as 3bus1, but with the micro-operation cycle times and latencies of Min2.

Prgm DAG	MAP Arch	No. of Valid Schds	No. of Optimal Schds	The Optimal Mkspn	*The MAP Heuristic*			*MAP with Lookahead*			
					Make-span	Close-ness	The Range	Make-span	Close-ness	The Range	New Schd?
BT3	Min1	640	24	1105nS	1185nS	92.76%	75%	1185nS	92.76%	75%	No
BT3.5	Min1	230400	512	1505nS	1585nS	94.68%	85.71%	1585nS	94.68%	85.71%	No
BT4	Min1	21964800	529920	1785nS	1885nS	94.4%	85.71%	1885nS	94.4%	85.71%	No
DD	Min1	42	2	1325nS	1325nS	100%	100%	1325nS	100%	100%	No
DM	Min1	310160	200	1905nS	1925nS	98.95%	98.11%	1905nS	100%	100%	Yes
SM	Min1	46574	24	2085nS	2245nS	92.33%	81.81%	2265nS	91.37%	79.55%	Yes
CC	Min1	4	2	735nS	735nS	100%	100%	735nS	100%	100%	No
CCL	Min1	4032	4	945nS	1015nS	92.59%	88.89%	1015nS	92.59%	88.89%	No
BT3	3bus1	640	72	1105nS	1105nS	100%	100%	1105nS	100%	100%	No
BT3.5	3bus1	230400	128	1355nS	1355nS	100%	100%	1355nS	100%	100%	No
BT4	3bus1	21964800	456960	1605nS	1605nS	100%	100%	1605nS	100%	100%	No
DD	3bus1	42	2	1225nS	1225nS	100%	100%	1225nS	100%	100%	No
DM	3bus1	310160	156	1645nS	1645nS	100%	100%	1645nS	100%	100%	No
SM	3bus1	46574	46	2005nS	2035nS	99%	97.67%	2005nS	100%	100%	Yes
CC	3bus1	4	2	735nS	735nS	100%	100%	735nS	100%	100%	No
CCL	3bus1	4032	18	835nS	835nS	100%	100%	835nS	100%	100%	No
BT3	Min2	640	32	930nS	930nS	100%	100%	930nS	100%	100%	No
BT3.5	Min2	230400	704	1230nS	1230nS	100%	100%	1230nS	100%	100%	No
BT4	Min2	21964800	768768	1500nS	1500nS	100%	100%	1500nS	100%	100%	No
DD	Min2	42	2	570nS	570nS	100%	100%	570nS	100%	100%	No
DM	Min2	310160	120	1250nS	1280nS	97.6%	92.5%	1250nS	100%	100%	Yes
SM	Min2	46574	2	1180nS	1200nS	98.3%	95.9%	1190nS	99.15%	97.96%	Yes
CC	Min2	4	2	400nS	400nS	100%	100%	400nS	100%	100%	No
CCL	Min2	4032	2	550nS	550nS	100%	100%	550nS	100%	100%	No
BT3	3bus2	640	32	920nS	920nS	100%	100%	920nS	100%	100%	No
BT3.5	3bus2	230400	704	1220nS	1220nS	100%	100%	1220nS	100%	100%	No
BT4	3bus2	21964800	2377728	1500nS	1500nS	100%	100%	1500nS	100%	100%	No
DD	3bus2	42	2	490nS	490nS	100%	100%	490nS	100%	100%	No
DM	3bus2	310160	1620	1230nS	1230nS	100%	100%	1230nS	100%	100%	Yes
SM	3bus2	46574	8	1160nS	1180nS	98.28%	96.01%	1160nS	100%	100%	Yes
CC	3bus2	4	2	400nS	400nS	100%	100%	400nS	100%	100%	No
CCL	3bus2	4032	12	550nS	550nS	100%	100%	550nS	100%	100%	No

Table 6–1: Measuring the optimality of the scheduling heuristics

The results for the MAP scheduling heuristic, both without and with instruction lookahead, are shown in Table 6–1. For each DAG, the number of valid schedules is recorded together with the optimal makespan for the given target architecture. The makespan generated by the heuristics together with its closeness to the optimum (recorded both as a percentage of the optimal (*Closeness*) and as a percentage of the difference between the best and worst makespans (within *The Range* – best being 100%, worst 0%)) are also included. It is assumed that there are a sufficient number of registers available to avoid code spilling. This would normally be determined at the register allocation phase of the compilation and is not considered here (see Chapter 7). If the lookahead heuristic generates a different schedule, this is indicated in the column "*New Schd?*".

The results look quite promising. In a majority of the cases for the **3bus1** and **3bus2** architectures, the MAP heuristic can find an optimal solution (only in the case of SM is instruction lookahead required, for both architectures, to reduce the makespan to optimum). However, the MAP scheduler does not do as well on the **Min1** architecture (for BT3, BT3.5, BT4, CCL, DM and SM). The reason for the poorer makespans is due to a bottleneck on the write-back bus. So significant is the effect of the bottleneck that even applying instruction lookahead, i.e. waiting until a higher priority instruction becomes ready rather than issuing the current one, has little effect. It turns out to be better in some cases to stall the issue unit for a much longer period of time than that assumed by the lookahead heuristic (of just the IICT), because this additional stall time would be hidden by the write-back bottleneck. The bottleneck can actually cause the lookahead heuristic to generate a schedule (e.g. for SM) whose makespan is worse than the one generated by the original MAP heuristic. The makespan would have been significantly better if it were not for the bottleneck (c.f. SM on Min2). Where the makespan is only slightly worse than the optimum, i.e. DM, the heuristic together with instruction lookahead is sufficient to find an optimal solution. In the case of the **Min2** architecture, BT3, BT3.5, BT4, and CCL are now optimal. This is because the relative delays of the microagents have reduced the bottleneck for the write-back bus. In the case of DM and SM, there is still interference between the instructions which result in sub-optimal executions. This instruction interference can be reduced by applying a post-pass re-ordering of the generated schedules.

6.6.1 Post-pass Optimisation for Instruction Interference

Instructions are said to interfere when a higher priority instruction's flow (i.e. execution) through the micronet is delayed by an instruction of a lower priority. For example, when an instruction is stalled waiting for a common resource, such as the write-back bus.

The MAP scheduler, which is mainly concerned with minimising the stall time of the issue unit, will generally choose to issue a lower priority instruction rather than wait for a higher one to become ready. However, the instruction lookahead heuristic tries to counterbalance this effect, albeit in a limited fashion.

As described in Chapter 5, the issue unit can only control the order in which operands are fetched (via the pre-issue conditions), i.e. the execution order of the micro-operations of microagents up to the execution stage. After this stage, the order in which instructions acquire successive microagents, especially common ones, may not necessarily be the same as the order in which the instructions were scheduled. This is due to multiple paths which allow instructions to race each other; the ability to skip stages; and varying stage delays, all of which are afforded by the micronet. In fact, it is difficult to predict how the microagents will be utilised as the schedule is being generated (i.e. on-the-fly), since a yet-to-be scheduled instruction could still determine whether an already scheduled one is serviced by a given microagent at a particular time. Therefore, any instruction interference optimisations can only be made after the initial schedule has been generated.

The only optimisation that can be made by the scheduler is to reorder the instructions. The post-pass heuristic, described in Algorithm 5, tries to ensure that instructions on critical paths are never delayed by those which are not. The heuristic uses the earlier instruction scheduling priorities and the schedule's "trace" information from the computational model to determine whether the issue order of two successive instruction should be swapped. Line 4 locates an instruction which is scheduled after one with a lower critical path priority. If the two instructions use a common microagent, in this case a write-back bus (line 5), the trace information is used to determine if the second, higher priority instruction is delayed by the first. This delay can easily be identified if the second instruction is stalled at its previous microagent (line 8). However, the heuristic (line 10) also assumes that if the second instruction requires the common microagent just as the first one finishes with it, then the control unit must have delayed (i.e. stalled) the issuing of the second instruction. The algorithm is applied to successive pairs of instructions in the schedule in a manner similar to the well known bubblesort algorithm. Although, this heuristic may increase the stall time of the issue unit, it has the overall effect of performing a restricted form of resource lookahead.

Algorithm 5 : ***Post-Pass Optimisation (reduce_interference())***

```
1   do
2      SWAP = NO;
       \* Assign InstA and InstB to the first two instructions in the schedule. *\
3      do
4         if ((SWAP == NO) and (InstA.ETC < InstB.ETC))
             \* Possible swap between InstA and InstB. *\
5            if (use_same_wbbus(InstA, InstB) == TRUE)
                \* Both instructions use the same write back bus. *\
6               InstA.et = time at which InstA relinquishes the write-back bus;
7               InstB.rt = time at which InstB requires the write-back bus;
8               if (InstA.et ≥ InstB.rt) SWAP = YES;
                   \* InstA delays InstB by the difference in these values. *\

9            else \* No swap required since instructions use different resources. *\
                \* Get the next pair of instructions in the schedule. *\

10        else \* Get the next pair of instructions in the schedule. *\
11     while not the end of the schedule;
12     if (swap == YES) simulate_schedule();
          \* Obtain the new schedule's trace info for next iteration. *\

13  while (swap == YES);
```

		The	*MAP with Lookahead (LA)*			*MAP with LA and Post-pass*		
Prgm DAG	MAP Arch	Optimal Mkspn	Make-span	Close-ness	The Range	Make-span	Close-ness	The Range
BT3	Min1	1105nS	1185nS	92.76%	75%	1105nS	100%	100%
BT3.5	Min1	1505nS	1585nS	94.68%	85.71%	1505nS	100%	100%
BT4	Min1	1785nS	1885nS	94.4%	85.71%	1785nS	100%	100%
DD	Min1	1325nS	1325nS	100%	100%	1325nS	100%	100%
DM	Min1	1905nS	1905nS	100%	100%	1905nS	100%	100%
SM	Min1	2085nS	2265nS	91.37%	79.55%	2085nS	100%	100%
CC	Min1	735nS	735nS	100%	100%	735nS	100%	100%
CCL	Min1	945nS	1015nS	92.59%	88.89%	965nS	97.88%	96.82%
BT3	3bus1	1105nS	1105nS	100%	100%	1105nS	100%	100%
BT3.5	3bus1	1355nS	1355nS	100%	100%	1355nS	100%	100%
BT4	3bus1	1605nS	1605nS	100%	100%	1605nS	100%	100%
DD	3bus1	1225nS	1225nS	100%	100%	1225nS	100%	100%
DM	3bus1	1645nS	1645nS	100%	100%	1645nS	100%	100%
SM	3bus1	2005nS	2005nS	100%	100%	2005nS	100%	100%
CC	3bus1	735nS	735nS	100%	100%	735nS	100%	100%
CCL	3bus1	835nS	835nS	100%	100%	835nS	100%	100%
BT3	Min2	930nS	930nS	100%	100%	930nS	100%	100%
BT3.5	Min2	1230nS	1230nS	100%	100%	1230nS	100%	100%
BT4	Min2	1500nS	1500nS	100%	100%	1500nS	100%	100%
DD	Min2	570nS	570nS	100%	100%	570nS	100%	100%
DM	Min2	1250nS	1250nS	100%	100%	1250nS	100%	100%
SM	Min2	1180nS	1190nS	99.15%	97.96%	1180nS	100%	100%
CC	Min2	400nS	400nS	100%	100%	400nS	100%	100%
CCL	Min2	550nS	550nS	100%	100%	550nS	100%	100%
BT3	3bus2	920nS	920nS	100%	100%	920nS	100%	100%
BT3.5	3bus2	1220nS	1220nS	100%	100%	1220nS	100%	100%
BT4	3bus2	1500nS	1500nS	100%	100%	1500nS	100%	100%
DD	3bus2	490nS	490nS	100%	100%	490nS	100%	100%
DM	3bus2	1230nS	1230nS	100%	100%	1230nS	100%	100%
SM	3bus2	1160nS	1160nS	100%	100%	1160nS	100%	100%
CC	3bus2	400nS	400nS	100%	100%	400nS	100%	100%
CCL	3bus2	550nS	550nS	100%	100%	550nS	100%	100%

Table 6–2: The effects of Post-pass optimisations on Instruction Lookahead schedules

The results of this optimisation, shown in Table 6–2, on the schedules generated by the lookahead heuristic are quite dramatic. All of the schedules except one (HTL on Min1, which has been significantly improved nevertheless) are now optimal, including the makespan for SM on Min1 which was made worse by instruction lookahead (see Table 6–1). The results also show that the post-pass heuristic does not adversely affect any of the schedules (even those which are already optimal).

The results of applying the post-pass heuristic directly to the schedules

Prgm DAG	MAP Arch	The Optimal Mkspn	*The MAP Heuristic* Make-span	Close-ness	The Range	*MAP with Post-pass* Make-span	Close-ness	The Range
DM	Min1	1905nS	1925nS	98.95%	98.11%	1925nS	98.95%	98.11%
DM	Min2	1250nS	1280nS	97.6%	92.5%	1280nS	97.6%	92.5%
DM	3bus2	1230nS	1230nS	100%	100%	1230nS	100%	100%
SM	Min1	2085nS	2245nS	92.33%	81.81%	2085nS	100%	100%
SM	Min2	1180nS	1200nS	98.3%	95.9%	1260nS	93.22%	83.67%
SM	3bus1	2005nS	2035nS	99%	97.67%	2035nS	99%	97.67%
SM	3bus2	1160nS	1180nS	98.28%	96.01%	1180nS	98.28%	96.01%

Table 6–3: The effects of Post-pass optimisation on MAP instruction schedules

produced without using instruction lookahead are shown in Table 6–3. In the cases of DM on Min1, Min2 and 3bus2, and SM on 3bus1 and 3bus2, there is no improvement. The makespan for SM on Min2 is actually worse, while for SM on Min1 it is now optimal. (Note that all of these schedules are optimal when both lookahead and post-pass are applied.) This does **not** mean that the post-pass heuristic will only work for schedules which can be improved by lookahead (emphasised by BT3, BT3.5 and BT4 on Min1). But rather, that the heuristic seems to give better results on those which are.

This post-pass heuristic can be applied initially to either the beginning (*forward post-pass*) or the end (*reverse post-pass*) of the instruction schedule generated by the first pass scheduler. The final schedules of the two approaches are identical, however reverse-postpass tends to attempt (to test for) more swaps.

6.6.2 Are These Schedules Really Optimal?

Remember that these schedules are only optimal with respect to the instruction costs which have been assumed. In practice, these schedules may not be optimal for a particular execution of the program for the reasons discussed earlier, i.e. the behaviour of the micronet is difficult to predict *a priori* and therefore instruction schedules are based on worst-case costs. One could even expect that each run of the program would have a different optimal schedule. Therefore it is impossible to determine how far from true optimality the schedules are, without in effect executing the actual instructions on the target architecture. This technique of scheduling through self-simulation has already been proposed when scheduling without a precise computational model [10]. The practicalities of such an approach are still open to question. Although the stability of the schedules in light of variance in the resource delays needs further study, this does not mean that good (at least comparable with synchronous systems) program executions cannot be achieved.

6.7 Open Problems

6.7.1 Instruction Execution Costs

The Scheduling Costs

	Worst-case Costs	Average-case Costs
Issue Cycle Time	3	2
ALU Instruction	4	3
Load Instruction	10	4
Store Instruction	6	3

The Schedule Based on Worst-Case Costs			**The Schedule Based on Average-case Costs**		
Instruction Schedule	Execution Time using worst-case run-time costs	Execution Time using average-case run-time costs	Instruction Schedule	Execution Time using worst-case run-time costs	Execution Time using average-case run-time costs
LD	0 - 10	0 - 4	ALU	0 - 4	0 - 3
ALU	3 - 7	2 - 5	LD	3 - 13	2 - 6
ALU	6 - 11	4 - 7	ALU	6 - 10	4 - 7
ST	10 - 16	7 - 10	ST	9 - 19	6 - 9
ALU	13 - 17	9 - 12	ALU	16 - 20	8 - 11
Execution Times :	17	12		20	11

Figure 6–1: The makespans of schedules based on worst- and average-case run-time costs

In a micronet-based processor, the actual execution times of instructions cannot be accurately predicted at compile-time. Although the execution times of the same instruction might vary due to data-dependent delays, worst-case, average-case or even best-case figures for the execution cost can be found on which the schedules could be based. When producing static schedules, the compiler has to use the delays of the FMs and the question arises as to which of the sets of figures to use. Figure 6–1 illustrates the simplified schedules for the Hennessy Test (HT1) based on worst-case and average-case costs and figures for the execution times of the instructions based on actual worst-case and average-case delays at run-time for these schedules. The ratios of the delays for the two cases for the instructions realistically reflect actual behaviour for the asynchronous processor under study. The figures reveal that given these ratios, using a schedule based on worst-case costs is better in practice. Using this approach a heuristic will always try to schedule an instruction, if possible, only when its operands are guaranteed to be available, thereby minimising any stalls. Note also that the schedule's correctness is not affected by the changes in instruction costs. Furthermore, given that a program's critical path may change with different executions (due to different data sets) and that the schedule is generated once, the compiler's choice of which costs to use is important (e.g. for real-time programmers [133]). By basing the schedule on worst-case delays a lower bound on performance can be achieved.

6.7.2 Interaction Between Executing Instructions

While optimising the instruction schedule is more difficult than in synchronous processors for the reasons stated previously, other reasons contribute as well, such as the difficulty in predicting the global state of the micronet. In synchronous processors, the compiler can assume when scheduling a basic block that the datapath is idle and all of the resources are available. This is a consequence of the fact that in synchronous pipelines, an instruction never affects the execution of other instructions. This is not necessarily the case in a micronet, since the execution times of instructions might vary for the following reasons: only a partial ordering is employed between instructions (i.e. it is not necessary for the previous instructions to have completed their execution before successive ones); instructions compete for shared resources, e.g. the write-back bus; during execution instructions might interfere with each other. Therefore, the state of the resources at any particular time cannot be predicted accurately at compile-time. But this information is indeed available at run-time in the issue unit of the micronet. This could be used to dynamically tune (i.e. allow out-of-order instruction issues) the static schedule by the control unit. This requires identifying an instruction which can be executed immediately (easily achieved using the control acknowledgement signal scoreboarding mechanism), and checking that the instruction is independent of earlier ones in the instruction buffer. Although the latter may be expensive to perform, the task can be made simpler with assistance from the compiler by using a concurrency bit.

6.8 Conclusions

The micronet model exposes temporal and spatial concurrency in the datapath, with fine-grained resources now being visible to the compiler. This model subsumes the micropipeline model which only exploits temporal concurrency in the datapath and the scheduling methods described here can be equally applied to micropipeline-based processors.

Code scheduling (on ILP architectures) and machine-dependent optimisations have a significant impact on program performance. It is the task of the compiler to schedule instructions such that these resources are efficiently utilised. The instruction schedule is devised based on a (parameterised) computational model of the target architecture. For synchronous architectures the model is simple; in contrast, an asynchronous model is necessarily less accurate for the reasons discussed earlier. However, initial studies have shown that these factors do not significantly hinder a MAP compiler's ability to schedule code efficiently. Worst-case instruction execution times have been considered for the reasons described earlier and the resulting schedule is treated as a first pass one. The interference between the instructions can be reduced by applying post-pass optimisations. The instructions could then be dynamically reordered at run-time to fine-tune this schedule by taking advantage of actual run-time costs. Due to the asynchronous behaviour these instructions are issued as soon as possible, without the need for delays using NO-OP instructions. In conclusion, preliminary studies have shown that a micronet-based asynchronous processor architecture does present a suitable target for an ILP compiler.

Chapter 7

Conclusions and Future Work

7.1 A Summary

Traditionally, the sequencing of information within processor architectures has been synchronous – centrally controlled by a clock. This global clock places limits on future gains in performance which can potentially be achieved by improvements in implementation technology. This thesis has investigated the effects of relaxing the strict synchrony by distributing control within the processor architecture and also its impact on the overall system design. Micronets have been proposed as an efficient implementation of an asynchronous control paradigm for processor architectures and their effect on system performance has been explored on three fronts. Firstly, with respect to an instruction set, the execution time of individual instructions were compared under the two control alternatives. A synchronous RISC architecture was transformed into a comparable self-timed one and simulation studies demonstrated improvements in the performance of the instruction set over the corresponding synchronous processor. Secondly, although improved performance through increased silicon utilisation within architectures which exploit instruction-level parallelism (ILP) in the form of pipelining is a key feature in processor designs, synchronous designs actually incur an increase in control complexity which adversely affects their efficiency. Based on an initial MAP design, a series of refinements have been made to the control framework which shows that the micronet approach is better able to exploit the available ILP amongst the functional units within processor architectures, and without significantly increasing control complexity. In micronets, not only can the handshake protocols be used to avoid hazards and minimise stalls, but the overheads due to asynchrony can also be hidden. Finally, although additional processor performance within the datapath has been exposed, whether or not the system benefits depends on a compiler's ability to exploit this improvement. An architecture needs to expose the available resource concurrency, while the compiler extracts the program parallelism (architecturally-independent) and maps it onto the former in such a manner as to maximise performance. Machine dependent optimisations and code scheduling (on ILP architectures) have a significant impact on the overall system performance. Performance gains obtained by RISC compilers have been due to the availability of accurate models of instruction behaviour on their target architectures. However, under asynchronous control, the resulting variable and non-predeterministic execution time of instructions due to data dependent operations does not seem to adversely effect the generation of good schedules. In conclusion, the adoption of micronets as an asynchronous distributed control paradigm can lead to a more efficient utilisation of functional units and thus improved system performance.

7.2 Effects on System Design

It is well known that the effective performance of a well integrated computer system is to a large measure determined by the synergy between the design of the processor architecture, the instruction set and the compiler. Therefore,

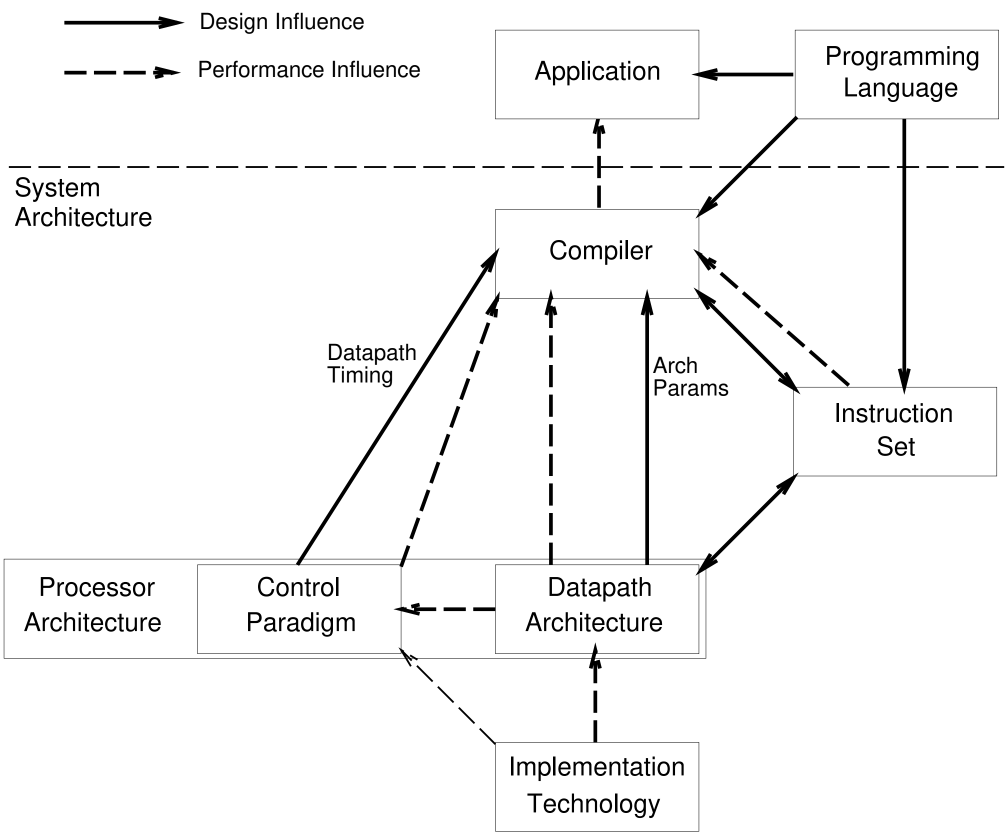


Figure 7–1: Influences within processor system architectures

the design of any such system should consider each of these areas and their relationship to each other. Furthermore, as this work has highlighted, another area (namely the control paradigm) also requires consideration (Figure 7–1):

The **Instruction Set** design is determined by the type of applications or programming languages for which the system is targeted, with RISC designers choosing to include only those instructions which are likely to be used frequently and whose exclusion would seriously degrade the system's performance [11]. Although the trend from sets containing a large number of complex instructions towards sets containing fewer less complex ones has led to simpler datapath and control, an improvement in overall system performance is still dependent on the compiler.

The **Compiler** is responsible for the construction of a (near) optimal (minimal code size or execution length) sequence of instructions which implements the target application program (written in a specific programming language). The advent of reduced instruction set architectures saw the accelerated development of optimising compiler techniques. These techniques have had a significant impact on system performance, thus making compilers, at least their back-ends, an integral part of a processor system. Code generation/scheduling and machine dependent optimisations require a detailed knowledge about each instruction's execution behaviour on the target architecture.

The **Datapath Architecture**, which is a collection of components (register bank, functional units, etc.) and their control signals and datapath interconnections (dedicated or shared), aims to implement the execution of each instruction as efficiently as possible which may also involve considering trade offs between power consumption, silicon area and performance. By streamlining the datapath of an architecture, its complexity can effectively be migrated to the optimising compiler. Details of the architecture become easier to make visible to the compiler and its computational model which reflects the behaviour of the architecture is now more tractable. The regular and determinate behaviour allows optimising techniques to be more effective.

The **Implementation Technology** (IT) has played a significant part in improving performance of processor architectures: transistorisation, various process technologies, scaling, fabrication techniques have all played their part. However, current advances in IC technology affect a synchronous control paradigm's ability to exploit the performance gains available (as discussed earlier in Chapter 2).

The **Control Paradigm** (CP), as the name suggests, is the mechanism by which the operation of the components within the datapath architecture are coordinated. Throughout the history of computer architecture, with a few exceptions, this has been synchronous where a global clock signal sequences actions and whose period is used to account for delays. With the majority of processor designs being based on a centralised synchronous control, the notion of a control paradigm has never been an issue.

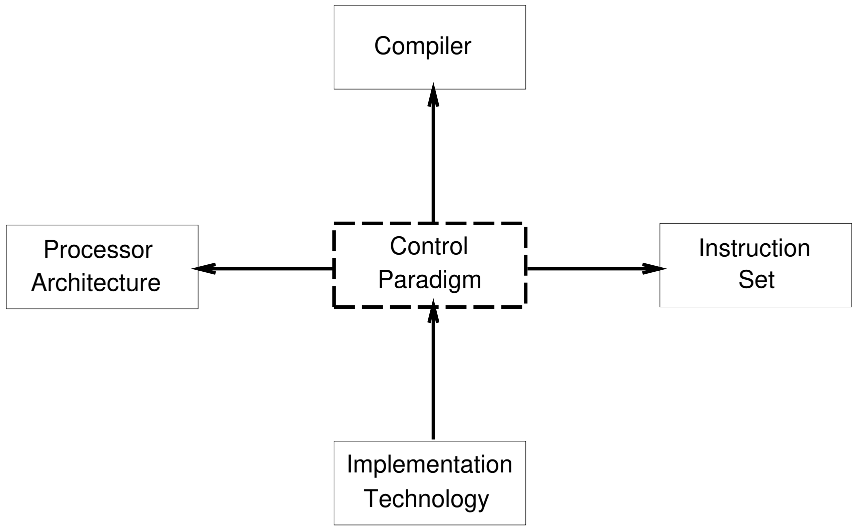


Figure 7–2: Previously implicit influences within system architectures

This thesis has explored an *asynchronous* control paradigm where the sequencing is decentralised and architectural components communicate using handshaking protocols. *Self-timed* control has been investigated together with its influence on the above areas and thus the overall effect on the performance of an integrated system. This work has not set out to find the best instruction set for asynchronous processors since it is felt that the CP does not significantly restrict the choice of instructions that could be included. In fact, an asynchronous CP is less restrictive since instruction execution times and design delay do not effect its correctness. However, some implementations of instructions, such as those which rely on the timing of operations or other instructions, may not be efficient or even possible.

On the other hand, the CP's influence on the datapath architecture is definitely more marked. While it is possible to implement a traditional synchronous architecture precisely in an asynchronous manner [137], one may find that the new design operates slower mainly due to the additional control required to force the design to operate or support certain features peculiar to the synchronous version. As described earlier in Chapter 2, the design goals under the two CPs are different, leading to possibly different implementations of the architectural components. The micronet approach with decentralised and distributed control, which does not preclude any particular architecture, does however lead to architectures composed of autonomous concurrently operating units.

From a purely performance point of view, for the modern RISC optimising compiler, the influence of an asynchronous control paradigm appears at first sight to be detrimental. The reason is the rôle played by the CP: synchronous control implies both an event ordering and timing which leads to predictable behaviour; asynchronous control implies only an event ordering. Without any timing information one might surmise that it becomes more difficult to optimise code schedules. In practice, preliminary results seem to imply that the compiler is not adversely affected by an asynchronous CP. Even though the schedules produced should be at least as good as those produced for a synchronous processor, the system should benefit from dynamic reordering to exploit further (run-time) performance.

The use of particular implementation technologies may also influence the choice of CP. Some types of MOS families may be considered well suited to self-timed circuit design, e.g. those which could use the precharge phase as a "spacer" between data values as an alternative method for hiding handshake overheads. Techniques such as differential cascade voltage switch level DCVSL [30] [69], Precharged CVSL [160] or domino CMOS logic [175] have already been used [98].

The behaviour of asynchronous processors is complex and their performance is difficult to predict. Discrete event simulations as described in this work offer a method for accurately measuring their performance. The model in Occam2 naturally captures the concurrency and asynchronous communication. This also allows the simulation to be parallelised to obtain reasonable run-times for large circuits and test programs. This is aided by the asynchronous nature of the underlying simulation algorithm itself [8]. Although there are numerous tools and techniques for the synthesis, verification and silicon compilation of self-timed circuits, tools for the development, evaluation and testing of self-timed (processor) systems [29] [51] are still lacking.

Given that micronets provide an efficient control framework for MAP systems many of these aspects are being addressed [92] [127] and could be investigated in more detail in future work.

7.3 On-Going and Future Work

7.3.1 Easing System Design

The distribution of control to the functional units improves performance by exploiting fine-grain concurrency and actual delays. The majority of control in MAP architectures is delegated to the interfaces of the functional units. The work in [126] has addressed the design of these control interfaces (the CMs) by introducing the idea of control constructs. These enable the efficient implementation of control interfaces which is crucial to the performance of the asynchronous processors. High-level descriptions of control constructs have been described in VHDL and a library of cells has been implemented in the Cadence Design Framework for automated synthesis [164]. Results from SPICE simulations for an add ALU operation have been presented which demonstrate the feasibility of distributing controls [4]. This work is an important step for the rapid prototyping of micronet-based asynchronous processors in a top-down fashion. The separation of timing and functionality enables truly modular designs, i.e. functional units can be modified without redesigning the rest of the system. Thanks to the micronet, the number and type of functional units can be changed, by simply specifying the behaviour of the control interface with respect to the rest of the system in terms of the control constructs. This enables the designer/computer architect to explore the architectural design space with ease, for example, determining the optimal number of functional units for a class of problems in the design of micronet-based superscalar architectures.

7.3.2 Extending the Micronet Architecture

Conditional Branching

The Fetch and Branch Unit (FBU) itself can be viewed as an instruction pre-processor, handling all PC related instructions. Its task is simply to supply the CU (and the execute stage) with, if possible, the correct stream of instructions. However, the implementation of (conditional) branch instructions is one of the hardest and most important problems to be dealt with in high performance pipelined processors. Branch instructions tend to interrupt the smooth flow of instructions through the datapath making the average instruction throughput rate much lower than the peak rate. For example, early studies for the pipelined MU5 computer showed that if branches occurred in only one out of ten instructions then performance would be reduced by a factor of three, unless precautions were taken [122]. The importance of dealing with the performance degradation has long been recognised [23]. Implementing branch instructions so that a branch transfer does not take effect until a fixed number of instructions after the branch are also executed can be used to reduce branch delay. This technique is commonly referred to as "delayed branching" and was used as early as 1952 in the Los Alamos MANIAC and more recently in early RISC processors such as IBM 801 [138], the Berkeley RISC I [136] and the Stanford MIPS [71]. Delayed branching is one of the simplest ways to optimise branches in synchronous architectures. However, a major limitation is the difficulty of filling the required number of delay slots determined by the time taken to resolve the branch condition [113]. While this number is fixed for a synchronous architecture, the number of instructions required to be fetched to hide the branch latency in an asynchronous datapath may be variable, depending not only on the execution cost but also the relative instruction fetch cost. Although this approach could be used for a specific MAP architectural design, as a general approach it is not viable. Thus, for micronet-based architectures, the preferred techniques might be ones which do not rely on fixed timing for their correct operation, such as branch prediction schemes [100] [153] or advanced branching mechanisms [132].

Out-of-Order Instruction Issue

Since the compiler may not be able to generate the best schedule, the CU may need to issue instructions out-of-order from the instruction buffer. This requires the identification an instruction which can be executed immediately (easy and cheap since the handshake mechanism with the functional units acts like a scoreboard), and checking that it is independent of the previous instructions in the buffer which might be expensive (dynamic register renaming) without the compiler's help [120].

Out-of-order instruction issue would allow the control unit to fine-tune the static instruction schedule to take advantage of variable instruction execution times. In the presence of out-of-order instruction issue (or out-of-order operand fetch), the issuing (and execution) of instructions is only limited by the availability of resources and operands. Micronets can therefore be viewed as a hybrid dataflow style of architecture, limited to the window of instructions available in the instruction buffer, without the bookkeeping costs associated with traditional dataflow architectures [61].

Exception Handling and Speculative Execution

Many synchronous processor architectures have been developed to exploit high degrees of ILP. Some of these processors dispatch multiple instructions from a conventional linear instruction stream to multiple functional units simultaneously and use mechanisms for out-order instruction issue and completion, branch prediction and speculative execution to remove the constraint of sequential instruction execution. The added complexity brought about by these mechanisms make it more difficult for the processor to maintain a precise system state after an exception occurs [81]. An exception is said to be precise if the saved process state corresponds with a sequential mode of program execution where one instruction completes before the next one begins. Many of the methods adopted by these synchronous processors for implementing precise interrupts [154] can be applied to MAP. For example, a history buffer (which is a first-in-first-out (FIFO) queue of all the instructions that are executing) can be used in the same way as it is in the MC88110 [40]. Alternatively, by introducing some processing (decision making) capabilities into the register bank, techniques equivalent to shadow registering [95] or the use of reorder buffers [154] could also be employed [155].

Although instructions may be fetched speculatively by the FBU in MAP, whether they should be executed speculatively is an architectural trade-off. Just as in synchronous designs, the techniques and hardware support for exception handling can be exploited to support speculative execution [40] [155] [167].

Extending Micronets to Implement Superscalar Architectures

The evolution of a synchronous scalar architecture into a superscalar one generally requires the duplication of the entire datapath. In MAP, this may not be necessary for a number of reasons: since the fetch and execute stages are decoupled, the effective instruction fetch rate may be sufficiently fast enough to mean that duplication of the fetch stage is not required; superscalar architectures exploit spatial parallelism and this is already achieved to a degree by a native/scalar micronet datapath; the natural extensibility of the micronet means that the incorporation of additional resources can be easily and efficiently exploited given a sufficiently fast enough instruction issue rate. Should this not be the case, the duplication of the instruction issue unit is possible (making the architecture superscalar) with more microagents to support concurrent operand fetching and the pre-issue conditions being modified to avoid new hazards and support out-of-order instruction issue. Due to the asynchronous behaviour, it would be inefficient to operate the instruction issue units in lock-step. The processor would now have to support complete dynamic instruction scheduling (out-of-order issue and out-of-order operand fetch). Johnson [81] provides a careful assessment of the complexity of the control logic involved in synchronous superscalar processors. The design and implementation of a superscalar micronet-based processor is currently being investigated [127].

Some Additional FU modifications

The designs of the functional units themselves may need to be modified to exploit the benefits of an asynchronous control paradigm or MAP architecture, e.g. average case delays. The Memory Unit (MU) services the load and store requests. While the simplest design option is to maintain the order in which the requests are serviced, in order to reduce the amount of time other functional units are stalled waiting for data, these load and store requests could be separated. Giving priority to load requests may reduce the data wait latency, although this requires the requests to be checked against any pending write request.

If the fetch stage has a sufficiently small delay, (i.e. there are no significant periods where the execute stage is starved of instructions), the FBU could be modified to allow it to be able to decode encoded instructions stored in memory, i.e. the unit can effectively be used as a pre-instruction-decode stage to speed up the instruction issue stage at the expense of a wider instruction buffer, increased fetch latency but smaller code size and perhaps lower power consumption [24] [48].

7.3.3 Parallelising Compilers for a Superscalar MAP

Although, Instruction-Level Parallelism (ILP) has been exploited by high performance uniprocessors for the past 30 years, the 1980s saw it play a much more significant rôle in computer design [94] [139]. ILP consists of a number of processor and compiler design techniques which are generally transparent to the user. Certain functions must be performed if a sequential program is to be executed in an ILP fashion: the program must be analysed to determine the type of dependencies between instructions and when these will be resolved; scheduling and register allocation must be performed; often operations must be executed speculatively, which in turn requires branch prediction. A number of design choices exist as to whether these functions are supported in the compiler or run-time hardware. Future MAP research should attempt to answer these questions.

Since a formidable amount of work has been done in the traditional ILP field [139], future work regarding the use of micronets may only need to consider the effects of an asynchronous control paradigm on ILP techniques (e.g. [17]). Although, some work has been done with List Scheduling heuristics, this approach may not produce the best results. Other interesting questions also arise, such as: with out-of-order instruction issue, how much work needs to be (and can be) done by the compiler? In practice, how much variance in instruction execution times should be expected in typical programs [53]? Also, how feasible is it to develop one efficient compiler for a family of MAP architectures?

7.4 Discussion

The emergence of VLSI technology, together with the maturing of optimising compiler techniques, aided the development of early RISC architectures [71] [86] [136]. Their primary concern was the efficient usage of expensive silicon real estate, and careful consideration was given to the design of the instruction set architecture [102]. There have been two orthogonal trends in the evolution of synchronous processor architectures [84]: the deeply-pipelined architectures [118], i.e. ones which exploit temporal parallelism, and superscalar architectures which exploit spatial parallelism [40] ([44] is an example which exploits both). Both these classes have benefited from improvements in technology and the resulting faster clock frequencies. But these improvements have been sustained at a high price in terms of clock distribution, power consumption, and design complexity [42]. Furthermore, significant additional control costs are incurred in exploiting ILP in both cases.

Micronets offer an alternative model for the design of future processor architectures. Whereas the original RISC ideal was the efficient usage of the silicon space by identifying the critical resources, a micronet is essentially concerned with their efficient utilisation over time. This is achieved in two ways: by removing the clock, and distributing control to the resources; and viewing the datapath not as a linear pipeline, but as a network of communicating resources. Micronets are able to efficiently (the overheads due to asynchrony being hidden) exploit fine-grain ILP without the additional control costs (the protocol also implements a scoreboarding and hazard avoidance mechanisms).

The asynchronous and distributed nature of the control in micronets allows the processor to be easily extended with little effect on the rest of the design. For a given class of problems, the designer is able to easily explore the architectural design space more accurately by adding critical resources. This can be naturally extended to superscalar architectures by increasing the number of issue units. (Synchronous superscalar architectures replicate entire datapaths.) The same scoreboarding mechanism is shared between the issue units for determining the global state of the datapath.

7.5 Conclusions

This thesis has highlighted the increasing inefficiencies due to the clock and centralised control in synchronous designs. Many of these problems can be avoided by using self-timed circuits and a method for converting synchronous pipelines to the self-timed equivalents has been outlined. This has been generalised to a novel asynchronous control technique, known as *Micronets*, for decentralising controls in asynchronous processor architectures. Micronets are viewed as a network of communicating functional units, which expose fine-grain concurrency between instructions.

This work has investigated the effect of removing synchrony in processor design and the consequent influences of an asynchronous control paradigm on the design and performances of RISC processor architectures for exploiting fine-grained ILP. It has been demonstrated that for a RISC architecture, the instruction execution of a self-timed design is able to exploit actual run-times. The advantages of an asynchronous control go even further, in being able to support instruction level concurrency. A Micronet-based Asynchronous Processor (MAP) architecture (which is effectively a variable length multiple-pipelined datapath) has been designed to efficiently exploit instruction-level parallelism and the nature of control for such an architecture has also been outlined. It has been demonstrated that four-phase handshaking protocols enable the implementation of highly concurrent structures and in most cases the overheads can be hidden. Just as importantly, these protocols are used to efficiently avoid datapath hazards. By using the self-timed design paradigm to the decentralised control, the control mechanisms in MAP are distributed amongst its functional units which allows the exploitation of a finer grain of ILP than previously possible. Improved architectural performance comes from being able to exploit both the actual run-time delays of the microagents and their concurrent operation. Some of the issues relating to micronets as targets for parallelising compilers have been discussed. Initial work has also confirmed the suitability of the asynchronous processor as a good target for these compilers. The modular nature of micronets eases modification and empowers the computer architect with finer control in the design, for example, of superscalar architectures. Finally, the micronet model considers the interactions between the underlying implementation technology, the architecture and the compiler, and underlines the integrated approach to system design.

Appendix A

Glossary

Actual (Program) Execution Time – The time between the issuing of an instruction (or start of a program) and the completion of all actions associated with that instruction (or program).

Asynchronous – An asynchronous circuit is an 'unclocked' circuit, i.e. a circuit which does not rely on global synchronisation by an external clock signal. Asynchrony implies the absence of any timing bounds on the operation of a circuit (whose duration may be subject to many uncontrolled factors).

Delay Insensitive – A circuit is delay-insensitive if its correct operation is independent of any assumptions about the delays of the individual components or wires in the circuit except that those delays be finite, c.f. *speed-independent*.

Equipotential Region – An equipotential region is a portion of a circuit within which propagation delays in wires are considered to be negligible. The smaller the area of the region, the more validity this assumption has in practice.

Fetch Cycle Time – The time between the Control (Execute) Unit requesting the next instruction from the instruction cache or memory and receiving it.

Instruction Cycle Time (ICT) – The execution time of a particular instruction as seen by the Control Unit. It is measured as the time between instruction issues of the same type.

Instruction Issue Time (IIT) – The time taken to issue an instruction. This constitutes just half of the four phase protocol and represents the time between decoding and issuing the instruction. (In synchronous designs, this would be the decode cycle with operand fetch occurring either concurrently or afterwards). In MAP, the fetching of operands is considered to be part of the instruction's execution. This is because the register bank is also treated as a functional unit or resource from which required operands may be unavailable.

Instruction Issue Cycle Time – The time between the issue of any two successive instructions. This is the time to complete the four-phase handshaking protocol and is therefore limited by the handshake cycle time of the slowest common control signal or IIT.

Isochronic Fork – A fork or branch of a wire in a circuit is considered to be isochronic if the difference in the propagation delays between branches is negligible. This is obviously the case if all branches of the fork are contained in an *equipotential* region.

Micronet – A *micronet* is a network of pipelines (micropaths), with (selected) stages of different pipelines being able to communicate with each other. This enables the exploitation of both spatial and temporal concurrency between instructions [4] (in contrast, a micropipeline only exploits temporal parallelism [6]).

Micropath – A *micropath* is a pipeline or sequence of *microagents*, and in turn, a microagent performs either a *communicating* or a *functional micro-operation*. A functional microagent (FM) communicates with other FMs through their respective communicating microagents (CM).

Micropipeline – A *micropipeline* is a self-timed, event-driven, elastic pipeline whose stages operate asynchronously and communicate using the two-phase bundled data protocol [158].

Self-Clocked – Self-clocked circuits are *self-timed* designs that are implemented using a hidden internal clock within an *equipotential region*. Although internally they are composed of clocked synchronous elements, self-clocked circuits retain an external asynchronous interface.

Self-Timed – Self-timed circuits use asynchronous initiation and completion (or request/acknowledge) signals. The class of self-timed circuits includes all *delay-insensitive*, *speed-independent* and *self-clocked* circuits.

Speed Independent – A circuit is said to be speed-independent if its correct operation is independent of the delays in the individual components of the circuit. It is assumed that there is no propagation delay associated with the wires of the circuit, c.f. *delay-insensitive*.

Appendix B

The PEPSÉ Simulator

B.1 The Simulation Algorithm in OCCAM2

```
{{{  PROC elsa.platform
PROC elsa.platform(CHAN OF ANY tty,
                   []CHAN OF INT::[]INT in,out,
                   VAL INT function.delay)
  -- Basic structure for the simulation platform.
  -- Folders marked with ** require modifications when customising.

  {{{  process runtime parameters **
  VAL INT max.input.width IS elsa.tuple.len.default+2:
  VAL INT max.output.width IS elsa.tuple.len.default+2:
  -- elsa.tuple.len.default is a constant currently set to 4. This
  -- is length of tuple with only one state value. Here, the input
  -- and output buffers will be defined to hold tuples with up to
  -- 3 state values.
  }}}

  {{{  variables
  [no.inputs][max.input.width] INT ipdata:
  [no.outputs][max.output.width] INT opdata:
                              -- Buffers for inputs and outputs.
  }}}
```

```
{{{  PROC function **
PROC function([][]INT istates,ostates)
-- This is the procedure which evaluates the output states given
-- the current inputs.
}}}

SEQ
  {{{  initialisation
  -- Set default values for flags
  }}}

  {{{  initialise input and output buffers
  PAR
    PAR i=0 FOR no.inputs
      PAR j=0 FOR max.input.width
          ipdata[i][j]:=0

    SEQ
                          -- Each output set to initial values
      PAR i=0 FOR no.outputs
        SEQ
          opdata[i][elsa.tup.len]:= elsa.tuple.len.default
          PAR j=1 FOR max.output.width-1
             opdata[i][j]:= tristate   -- initial state values.
          opdata[i][elsa.start.time]:= 0
          opdata[i][elsa.end.time]:= function.delay
  }}}

  {{{  send initial output tuples
  PAR i=0 FOR no.outputs
      out[i] ! opdata[i][elsa.tup.len]::opdata[i]
  }}}

  WHILE NOT finished.sim
    SEQ
      {{{  fetch necessary inputs
      PAR i=0 FOR no.inputs
        IF
          (ipdata[i][elsa.start.time]=ipdata[i][elsa.end.time])
            in[i] ? tuple.length::[ipdata[i]
                    FROM 0 FOR tuple.length]
          TRUE
```

```
      SKIP
}}}

{{{  execute function **
function(ipdata,opdata) -- Behavioural model of Object.
}}}

{{{  determine OUTPUT start time
PAR i=0 FOR no.outputs
    opdata[i][elsa.start.time] :=
    ipdata[0][elsa.start.time]+function.delay
}}}

{{{  determine OUTPUT end time
minimum.end.time :=max.sim.time
SEQ i=0 FOR no.inputs
  IF
    (minimum.end.time>ipdata[i][elsa.end.time])
      minimum.end.time := ipdata[i][elsa.end.time]
    TRUE
      SKIP

PAR i=0 FOR no.outputs
  opdata[i][elsa.end.time] :=
                minimum.end.time + function.delay
}}}

{{{  send outputs
PAR i=0 FOR no.outputs
  IF
    (max.sim.time > ipdata[i][elsa.end.time])
      out[i] ! opdata[i][elsa.tup.len]::
      [opdata[i] FROM 0 FOR opdata[i][elsa.tup.len]]
    TRUE
      SKIP
}}}

{{{  update simulation time
PAR i=0 FOR no.inputs
  ipdata[i][elsa.start.time] := minimum.end.time
}}}
```

```
        {{{  Simulation Complete ?
        IF
          (ipdata[0][elsa.start.time] >= max.sim.time)
            finished.sim := TRUE
          TRUE
            SKIP
        }}}

    {{{  Sink irrelevant inputs
    SEQ i=0 FOR no.inputs
      WHILE (max.sim.time > ipdata[i][elsa.end.time])
        in[i] ? tuple.length::[ipdata[i] FROM 0 FOR tuple.length]
    }}}
:
}}}
```

Appendix C

The MAP Test Programs

```
{{{  Instruction Test code
  {{{  Program - Load Test
  -- instruction format  <opcode,Rx,Ry,Rz,condflg,timestampflg>
  -- remember to initialise reg[i] = i
  instr[0] :=[ld,0,0,1,false,true]
  instr[1] :=[ld,0,0,2,false,true]
  instr[2] :=[ld,0,0,3,false,true]
  instr[3] :=[ld,0,0,4,false,true]
  instr[4] :=[ld,0,0,5,false,true]
  instr[5] :=[ld,0,0,6,false,true]
  instr[6] :=[ld,0,0,7,false,true]
  instr[7] :=[time,1,2,2,false,false]
  instr[8] :=[jmp,8,0,0,false,true]
  }}}

  {{{  Program - Store Test
  -- instruction format  <opcode,Rx,Ry,Rz,condflg,timestampflg>
  -- remember to initialise reg[i] = i
  instr[0] :=[st,0,1,1,false,true]
  instr[1] :=[st,2,0,2,false,true]
  instr[2] :=[st,0,3,3,false,true]
  instr[3] :=[st,4,0,4,false,true]
  instr[4] :=[st,0,5,5,false,true]
  instr[5] :=[st,6,0,6,false,true]
  instr[6] :=[st,0,0,7,false,true]
  instr[7] :=[time,1,2,2,false,false]
  instr[8] :=[jmp,8,0,0,false,true]
  }}}
```

```
  {{{  Program - Alu Test
  -- instruction format  <opcode,Rx,Ry,Rz,condflg,timestampflg>
  -- remember to initialise reg[i] = i
  instr[0] :=[add,0,0,1,false,true]
  instr[1] :=[add,0,0,2,false,true]
  instr[2] :=[add,0,0,3,false,true]
  instr[3] :=[add,0,0,4,false,true]
  instr[4] :=[add,0,0,5,false,true]
  instr[5] :=[add,0,0,6,false,true]
  instr[6] :=[add,0,0,7,false,true]
  instr[7] :=[time,1,2,2,false,false]
  instr[8] :=[jmp,8,0,0,false,true]
  }}}

  {{{  Program - Hennessy Test
  -- instruction format  <opcode,Rx,Ry,Rz,condflg,timestampflg>
  -- x[i] := k + x[j]; x addr in R0, (1,R1),(i,R2),(j,R3),(k,R4),(Xj,R5),(Xi,R7)
  instr[0] :=[ld,  0,3,5,false,true]
  instr[1] :=[add, 1,3,3,false,true]
  instr[2] :=[add, 5,4,7,false,true]
  instr[3] :=[st,  0,2,7,false,true]
  instr[4] :=[add, 1,2,2,false,true]
  instr[5] :=[time,0,0,0,false,true]
  instr[6] :=[jmp, 0,6,0,false,true]
  }}}
}}}
```

Appendix D

Published Papers

The copyright on each of the following papers has been transferred to the Elsevier Science Publishers and the IEEE Computer Society Press (as indicated), which have granted to the authors the right to republish without specific permission.

D.1 Instruction-level Parallelism in Asynchronous Processor Architectures

Title:	Instruction-level parallelism in asynchronous processor architectures.
Authors:	D. K. Arvind and **V. E. F. Rebello**.
Presented at:	The 3rd International Workshop on Algorithms and Parallel VLSI Architectures.
Place:	Leuven, Belgium.
Date:	29^{th} – 31^{st} August 1994.
Publisher:	Elsevier Science Publishers.

INSTRUCTION-LEVEL PARALLELISM IN ASYNCHRONOUS PROCESSOR ARCHITECTURES

D. K. ARVIND and V. E. F. REBELLO
Department of Computer Science, The University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ, Scotland, U. K.
{dka,vefr}@dcs.ed.ac.uk

ABSTRACT. The Micronet-based Asynchronous Processor (MAP) is a family of processor architectures based on the micronet model of asynchronous control. Micronets distribute the control amongst the functional units which enables the exploitation of fine-grained concurrency, both between and within program instructions. This paper introduces the micronet model and evaluates the performance of micronet-based datapaths using behavioural simulations.

KEYWORDS. Instruction-level parallelism (ILP), asynchronous processor architecture, self-timed design.

1 INTRODUCTION

Centralised controls have been traditionally used to correctly sequence information within processor architectures. However, the ability to sustain this design style is under pressure from a number of directions [6]. This paper examines the effect of relaxing this strict synchrony on the design and performance of processor architectures. The reasons are the following. The the clock frequency of a synchronous processor is determined *a priori* by the speed of its slowest component (which takes into account worst-case timings for execution and propagation for pessimistic operating conditions). In contrast, the performance of an asynchronous processor is determined by actual operational timing characteristics of individual components (effectively the average delays), and overheads due to asynchronous controls. Secondly, an important consequence of asynchronous controls is the ability to exploit fine-grained Instruction-level Parallelism (ILP), and this is explored in greater detail in the rest of this paper.

ILP can be achieved either by issuing several independent instructions per cycle as in superscalar or VLIW architectures, or by issuing an instruction every cycle as in a pipelined

Published in the Proceedings of the 3rd International Workshop on Algorithms and Parallel VLSI Architectures, pp 203-215, Leuven, Belgium, August 1994.

architecture where the cycle time is shorter than the critical path of the individual operations [5]. This work concentrates on the design and evaluation of asynchronous pipelines for exploiting ILP, as a number of control issues resulting from data and structural dependencies between instructions have to be resolved efficiently.

A few asynchronous processors have recently been proposed [3, 8, 9]. These designs are based on a single micropipeline datapath [10]. One disadvantage of viewing a datapath as a linear sequence of stages is that, in general, only one of the functional units will be active in any cycle. Pipelining the functional units themselves is expensive both in terms of additional hardware and the resulting increase in latency.

We introduce an alternative model for an asynchronous datapath called a *micronet*. This is a network of elastic pipelines in which individual stages of the pipelines have concurrent operations, and stages of different pipelines can communicate with each other asynchronously. This allows for a greater degree of fine-grained concurrency to be exploited, which would otherwise be quite expensive to achieve in an equivalent synchronous datapath.

2 MICRONETS AND ASYNCHRONOUS ARCHITECTURES

Micronets are a generalisation of Sutherland's micropipeline [10], which dynamically control which stages communicate with each other. Thus micronets can be viewed not just as a pipeline but rather as a network of communicating stages. The operations of each of the stages are further exposed in the form of *microagents* which operate concurrently and communicate asynchronously with microagents in other stages. Each program instruction spends time only in the relevant stages and for just as long as is necessary. This is in contrast with synchronous datapaths in which the centralised control forces each instruction to go through all the stages, regardless of the need to do so (in effect a single pipeline). Furthermore, the microagents within a stage might operate on different program instructions concurrently.

Micronets are controlled at two levels: the data transfer between microagents is controlled locally, whereas the type of operation carried out by a microagent (called a microoperation) and the destination of its result is controlled by the sequencer or by other microagents. Microagents can communicate either across dedicated lines or via shared buses where arbitration is provided either by the sequencer or some other decentralised mechanism such as a token ring.

Data dependencies in synchronous pipelines are resolved by using either hardware or software interlocks [4], which increases the complexity of the controls. Micronets use their handshaking mechanisms together with simple register locking to achieve the same effect, but with trivial hardware overheads. In synchronous designs the structural hazards are normally avoided in hardware by using a scoreboarding mechanism. In micronets this is provided by existing handshaking protocols. Out-of-order instruction completion can be supported in synchronous designs, but at a non-trivial cost. Micronets are able to relax the strict ordering of instruction completions and thereby further exploit ILP. The result is to effectively increase the utilisation of the functional units by reducing their idle times or stalls. Better program performances can be achieved by exploiting both ILP and actual instruction execution times.

2.1 Asynchronous Architectures

Figures 1-3 illustrate micronet models of a generic asynchronous RISC datapath. The intention is not to focus on the functional units themselves but rather on their asynchronous control and investigate their effect on the performance. The number of units and their functionality may be changed without side-effects.

The architecture can be described as a network of microagents (denoted by solid boxes) which are connected via ports. The microagents which are labelled in the figures, called Functional Microagents (FMs), perform microoperations which are typical of a datapath. On each of their ports are Communicating Microagents (CMs) which are responsible for asynchronous communications between FMs and the rest of the micronet. The FMs are effectively isolated and only communicate through their CMs, and can therefore be modified without affecting the rest of the micronet.

2.2 Measuring Performance

We next introduce a few metrics for measuring improvements due to the distribution of control. There are two principal characteristics which affect performance - the microoperation latency (the time between initiating the operation and the result being available), and the microoperation cycle time (the minimum time between successive initiations of the same operation, i.e. throughput). The metrics defined for MAP are as follows:

Minimum Datapath Latency (MDL) - The time between asserting the control signals (i.e. initiating instruction issue) and receiving the final acknowledgement of the instruction's completion.

Instruction Cycle Time (ICT) - The time between two identical instruction issues once that instruction's pipeline is full. In asynchronous pipelines which usually have non-uniform stage delays, the time between successive instruction issues is influenced by the slowest stage *currently* active in the pipe.

Program Execution Time (PET) - The actual execution time of the program.

A more detailed exposition of performance-related issues is presented in [1].

To study the effectiveness of the micronets, it is sufficient to focus on the LD, ST, and ALU instructions. Five simple test programs were devised to exercise the design. The Alu, Load and Store test programs measure the maximum attainable utilisation of their respective FMs. Each of these programs contain a number of identical instructions, such that only structural dependencies exist between instructions (in effect setting up a static pipeline or a fixed path through a network of components). The number of instructions in the test programs are sufficient to fill the pipeline, i.e. enough instructions exist for the Control Unit (CU) to achieve a steady issue rate. The Hennessy Test (HT1) consists of a mix of the previously-mentioned instructions but without any data dependencies, which exercises the spatial concurrency and out-of-order completion, both of which are provided by the micronet, for a particular schedule devised by the compiler. HT2 is a variant of HT1, with data dependencies, which exercises the data forwarding mechanism as well. This program represents a "typical" basic block of compiled code (actually a line of code in C from [4]).

To facilitate the simulation of instruction sequences within reasonable run-times and without sacrificing accuracy, the timing characteristics of the architecture (in 1.5 μm CMOS) were extracted from a post-layout simulation tool within a commercial VLSI design package called SOLO 1400 [2] and incorporated into a mixed-level (mainly register-transfer level) model. The processor was described in Occam2 and simulated on a parallel asynchronous event-driven simulation platform, on a transputer-based ME*i*KO Computing Surface.

3 REFINEMENTS

The following sections discuss a number of refinements which were made in three stages to the base design as shown in Figure 1. This highlights the ease with which the micronet model can efficiently exploit ILP and without the difficulties normally encountered in synchronous datapath design, such as implementing hazard avoidance, data-forwarding or balanced pipeline-stage design.

The processor design as illustrated in Figure 1 only exploits the actual execution timings of microoperations (Stage 1), whereas later designs exploit both this property and the available concurrency between the microoperations of different instructions. The execution of each instruction requires a predetermined set of microoperations, each initiated by signals from the CU. These are four-phased controls whose acknowledgement signals are used as status flags for mimicing a scoreboarding mechanism. In general, the microoperations for an instruction are initiated as soon as possible by asserting the necessary control signals. The receipt of an acknowledgement confirms that the associated microoperation has begun and the initiating control signal is de-asserted. The instruction is said to be issued once all the asserted control signals have been acknowledged, which then allows the next instruction issue to begin.

3.1 Stage 1

Figure 1 illustrates a naïve implementation of the datapath of an asynchronous processor, which does not as yet fully exploit the full repertoire of micronets. The control signals generated by the CU for Stage 1 are described in greater detail below:

Rx, (Ry) - This signal identifies the source register for the X (Y) Bus. The corresponding acknowledgement is asserted once the register has been accessed, and cleared once the data has been transferred to the operand fetch CM.

Rz - Same as above. The ST microoperation obtains the third operand over the Z Bus.

Rof - Same as above, but the value in the offset register is output onto the X bus.

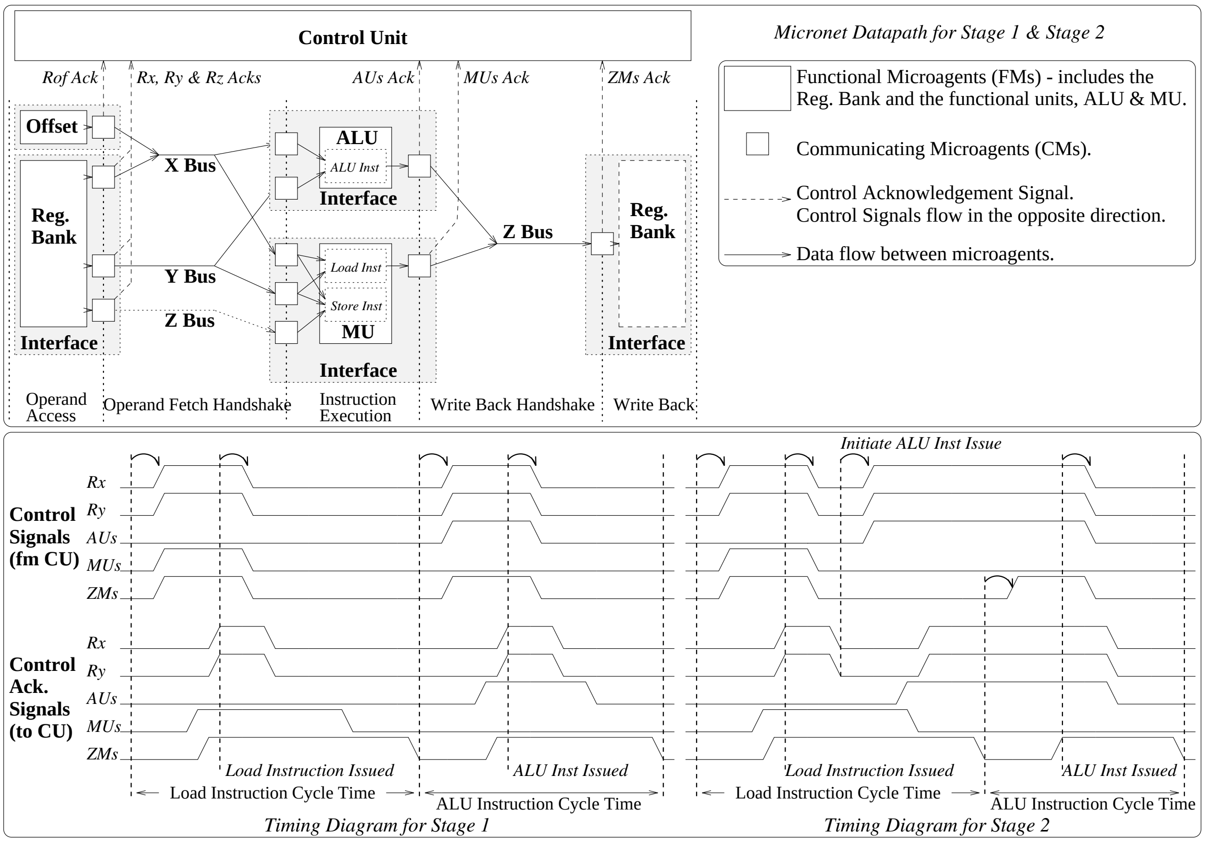


Figure 1: The micronet model of Stages 1 & 2

AUs - This signal identifies the next operation of the ALU. The corresponding acknowledgement is asserted when the interface is ready to fetch the ALU's operands from the registers and is cleared when it initiates the write-back handshake.

MUs - This signal identifies a load instruction to the MU and is asserted and cleared in the same manner as above. (Control signals for the other MU microoperations have been omitted for the sake of clarity).

ZMs - This signal identifies the destination register for data write-backs from the ALU or MU via the Z bus. The corresponding acknowledgement signal is asserted when the register is ready to receive data and cleared once the data has been written back.

In Stage 1, all the microoperations for a particular instruction are initiated together, and the next set cannot be initiated until the completion of the set of microoperations of the previous one. This effectively serialises the instruction execution, as illustrated in the timing diagram in Figure 1. In successive refinements the rôle of the CU is diminished by distributing the control of the micronet to local interfaces and microoperations are individually initiated as early as possible.

Instruction	Inst. Cycle Time (ICT)	Datapath Latency (MDL)
ALU	24nS	24nS
LD	43nS	43nS
ST	23nS	21nS

Table 1: Instruction Execution on Stage 1

In the base stage, the ICT is determined by the slowest control signal handshake since the next instruction issue cannot begin until all the previous handshakes have been completed. The results in Table 1 show that the ICT is equal to the MDL (except for the ST instruction), which is not surprising as instructions execute sequentially but only take as long as is necessary. The higher value for the ST instruction is due to a handshake delay, which in the LD instruction is hidden by the write-back stage. Although there is no explicit pipelining of the datapath, different phases of the handshaking may occur at the same time, e.g. a CM may initiate a handshake with another CM while completing one with its FM. As was expected the execution times of the test programs (Table 5) are the sum of their individual instruction execution times together with startup overheads.

3.2 Stage 2

The strict condition which was employed in Stage 1 for initiating a set of microoperations after the completion of the previous set is now relaxed. Furthermore, the CU can now assert any of the individual microoperations for an instruction asynchronously, where previously the set of microoperations for an instruction were initiated in unison. This allows microoperations relating to different instructions to overlap (Stage 2 in Figure 1). Note that a control signal which is related to an instruction can only be de-asserted once all of the relevant control signals have been acknowledged. The effect of relaxing this constraint is to introduce possible hazards and efficient mechanisms have been devised to avoid them. Fortunately, these hazard avoidance mechanisms are implicit in the orderings of the assertions of the control signals, known as the *pre-issue conditions* and these are discussed below:

Read-after-Write (RAW) - A register locking mechanism is implemented in the register bank without the CU having to keep track of the "locked" registers. The acknowledgement signal **ZMs** is asserted after the locking operation, and is de-asserted once the data is written back (signaling the unlocking of the register). By definition an instruction is issued once all the acknowledgements of the relevant microoperations have been received. This implies that the destination register of the previous instruction will have been locked before the CU initiates any of the current instruction's microoperations.

Write-after-Read (WAR) - This hazard is avoided without additional hardware overheads. When a register is used as both source and destination within the same instruction, then it is necessary to ensure that the source data is obtained before the register is locked, otherwise deadlock will occur. The CU stalls the assertion of **ZMs** until the source operand control signals **Rx** and **Ry** have been asserted.

Write-after-Write (WAW) - Although concurrent instruction execution can now take place, write-backs are still enforced in order. It is necessary to ensure that destination register has been locked, and that data is then written to its correct location. These conditions are met by simply preventing a functional unit (FU) from writing data back until the control signal from the CU has been de-asserted (an implicit go-write signal). This is sufficient since the control signals cannot be de-asserted before **ZMs** is asserted (see Figure 1). Note that if the CU attempts to lock a register which is already so, then the acknowledgement signal cannot be asserted and the current request will stall. This mechanism guarantees that write-backs to the same register occur in the correct order without stalling the instruction issue, and thereby allowing the instructions to execute concurrently with only the write-backs being sequential. The CDC6600 [11] used a similar go-write signal which sequentialised the execution of the offending instructions.

Operand Fetch - Simultaneous operand requests by FUs to the same Register Bank CM microoperation can lead to one of them acquiring the wrong operand. This can be avoided by the CU by delaying the assertion of the control signal to a FU until the previous FU has made its operand request(s) to the registers, i.e. until the acknowledgement signals of "operand fetch" microoperations have been de-asserted.

Bus Contention - Due to the mechanism to avoid WAW hazards only the Register Bank and either the ALU or Memory Unit can write onto the Z Bus simultaneously. Thus bus access is arbitrated by the CU through mutually-exclusive assertions of **Rz** and **ZMs**.

Instruction	Inst. Cycle Time (ICT)	Datapath Latency (MDL)
ALU	21nS	24nS
LD	42nS	43nS
ST	23nS	21nS

Table 2: Instruction Execution on Stage 2

The improvements in the instruction cycle times, as shown in Table 2, are small. This can be explained by the limited overlap between the operand access of the current instruction and the write-back of the previous one. In the design under consideration there can only be two program instructions active in the datapath simultaneously.

3.3 Stage 3

In Stage 3, the rôle of the CU is diminished further by distributing the control of the micronet to individual CMs. The CMs have been enhanced to more than just controlling local communications between FMs. They effectively buffer the initiations of the microoperations from the CU until their respective FMs are ready to perform. Also, the write-back to the register bank is no longer controlled by the CU, but directly by the CMs of the FMs which require the service, i.e. the write-back microoperation is initiated by the microoperations in the previous stage.

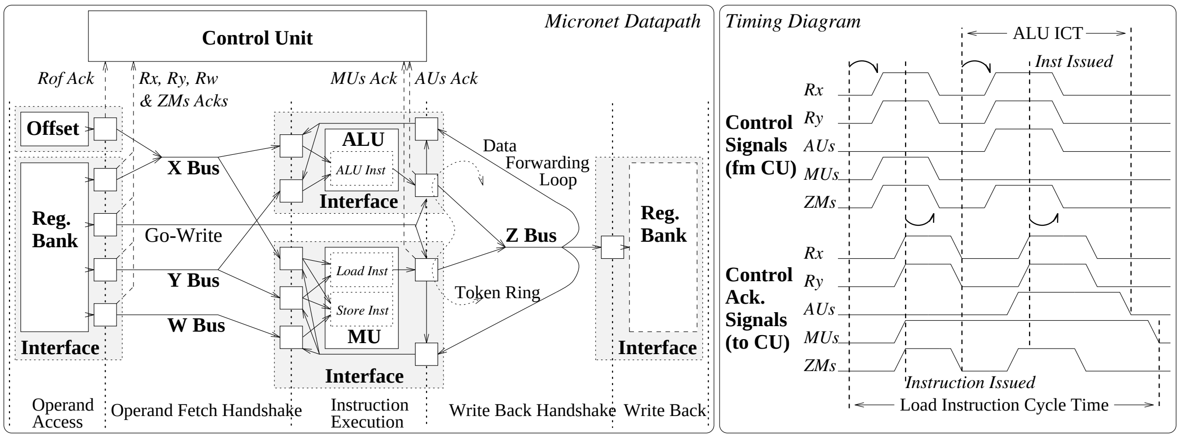


Figure 2: The micronet model of Stage 3

Enforcing write-backs in order restricts the degree of concurrency which can be exploited, especially when the FU executions times vary significantly. However supporting out-of-order completion of instructions in an asynchronous environment is more difficult than under synchronous control. Determining the precise order in which results will be available is virtually impossible since microoperation delays vary.

Out-of-order instruction completion is supported by tagging the write-back data with the address of the destination register. The CU cannot predict the write-back order, therefore a decentralised bus arbitration scheme as in a token ring is employed. The ring is distributed amongst the CMs and is very simple to implement in VLSI. However, the ring's cycle time will increase with the number of FMs, and might be infeasible for larger numbers.

With data transfer on the Z bus being tagged, CMs can identify and intercept operands for which it may be waiting. This mechanism is reminiscent of the IBM 360/91 common bus architecture [12]. Data-forwarding has been implemented by exploiting the feedback loops of the micronet. In the event of data forwarding, where data is routed directly to the CM of a waiting FM, the CM's previous request for that operand is in effect cancelled by initiating a separate handshake. This frees the corresponding "operand fetch" CM to service its next request. An alternative approach would be to implement operand bypassing, where the operand is fed back to the "operand fetch" microoperation. This avoids the need for data forwarding CMs and the cancel handshake. The dual rôle of the Z bus can no longer be supported due to the data-forwarding mechanism. A separate operand fetch bus (W bus) is used, thereby making the Z bus purely a write-back one (see Figure 2).

As a result of these modifications, the acknowledgements to the control signals and the pre-issue conditions have to be revised as shown below:

Rx, (Ry, Rw) - The acknowledgement is asserted by the CM of the register bank when the X (Y, W) bus operand fetch microoperation is ready, and de-asserted once the operand fetch handshake is in progress.

Rof - Same as above. Note that both the control signals **Rx** and **Rof** cannot be active simultaneously.

AUs, MUs - The acknowledgement is now asserted when the corresponding CMs are ready to fetch the operands from the registers and is cleared once the FM microoperation has completed.

ZMs - The acknowledgement signal is asserted when the CM is ready and de-asserted once the operation has been completed (i.e. the register has been locked).

RAW -The CU delays the assertion of the operand fetch control signals **Rx**, **Ry** and **Rw** until the previous **ZMs** control acknowledgement signal has been de-asserted, which indicates the locking of the previous destination register.

WAW - The mechanism is unchanged except that the go-write signal originates from the register interface and not the CU (i.e. the mechanism has now been decentralised).

Write-back Contention - This is prevented by the use of a token ring to arbitrate accesses to the write-back (Z) bus. Of course, this problem could be obviated by using dedicated buses for small number of FMs, but is impractical for designs with larger numbers.

Further concurrency is achieved by applying these pre-issue conditions only when necessary by explicitly checking register addresses for dependencies between successive instructions.

Instruction	Inst. Cycle Time (ICT)	Datapath Latency (MDL)
ALU	15nS	24nS
LD	38nS	43nS
ST	23nS	21nS

Table 3: Instruction Execution on Stage 3

We observe an improvement in the cycle times of instructions which require to write data back to the registers, such as the LD and ALU instructions, as shown in Table 3. This is due to the de-centralisation of the write-back control to the relevant CMs. These improvements are reflected in the shorter PETs for Load, Alu and HT1 test programs, as shown in Table 5. Columns "HT2" and "HT2(DF)" refer to the cases without and with data-forwarding, respectively.

3.4 Stage 4

In this final stage, both the assertion and de-assertion of the control signals now occur independently of each other. The states of the FU acknowledgements no longer represent the activity of their FMs, but rather that of their operand-fetch CMs. All of this further increases the concurrency between microoperations which makes possible the exploitation of fine-grained concurrency between instructions.

The ICT value for the LD instruction in Table 4 is the best attainable as it represents the MU delay for the operation. These figures show that the micronet can exploit the

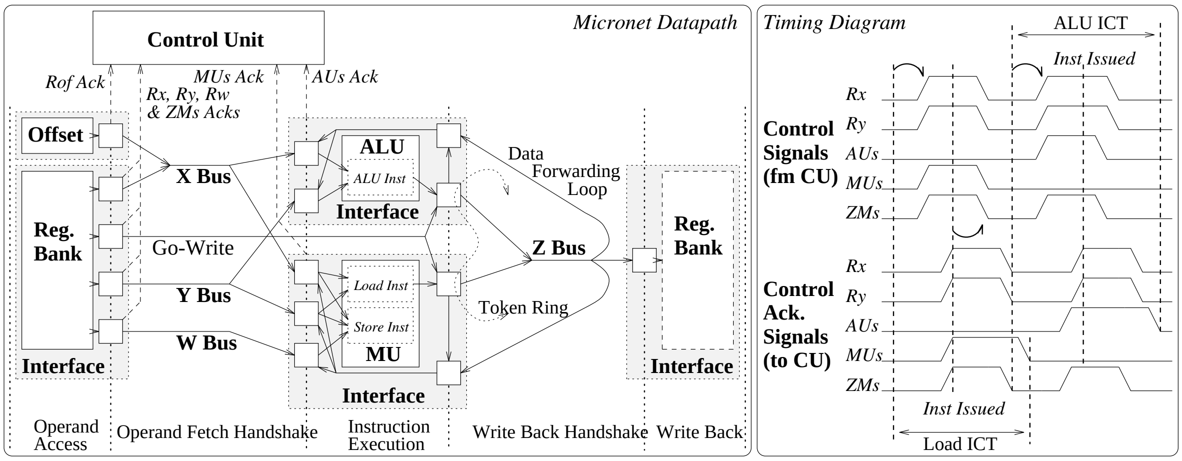


Figure 3: The micronet model of Stage 4

Instruction	Inst. Cycle Time (ICT)	Datapath Latency (MDPL)
ALU	12nS	24nS
LD	23nS	43nS
ST	12nS	21nS

Table 4: Instruction Execution on Stage 4

actual operational cost and effectively hide the overheads of self-timed design. The ICTs for the ALU and ST instructions are limited by their operand fetch cycle times. The overall improvements in the program execution times in Stage 4 over Stage 1 for the first three test programs (shown in Table 5 and Figure 4) are due to improvements in temporal concurrency due to the pipelining of the datapath. The actual speedup which is achieved is less than the maximum attainable improvement (the ratio of the ICTs in Tables 1 and 4), due to the MDL and the startup overheads (for longer test programs the speed-up will approach this maximum value). The speed-up for HT1 is due in part to pipelining of the instructions as observed in the other test programs, but also due to additional spatial concurrency due to the overlapping of different instructions in the same stage of the micronet. This further improvement is still significant (approximately 17% in this example) given that both successive instruction operand fetches and write-backs are effectively forced to take place sequentially due to resource constraints. (In fact, since these delays are larger than

PET	Alu Test	Load Test	Store Test	HT1	HT2	HT2(DF)
Stage 1	175nS	308nS	164nS	143nS	143nS	-
Stage 2	157nS	302nS	165nS	119nS	119nS	-
Stage 3	121nS	280nS	165nS	83nS	97nS	91nS
Stage 4	103nS	188nS	98nS	79nS	-	91nS
Effective Speed Up	1.75	1.66	1.71	1.89	-	1.62

Table 5: Execution Times of the Test Programs

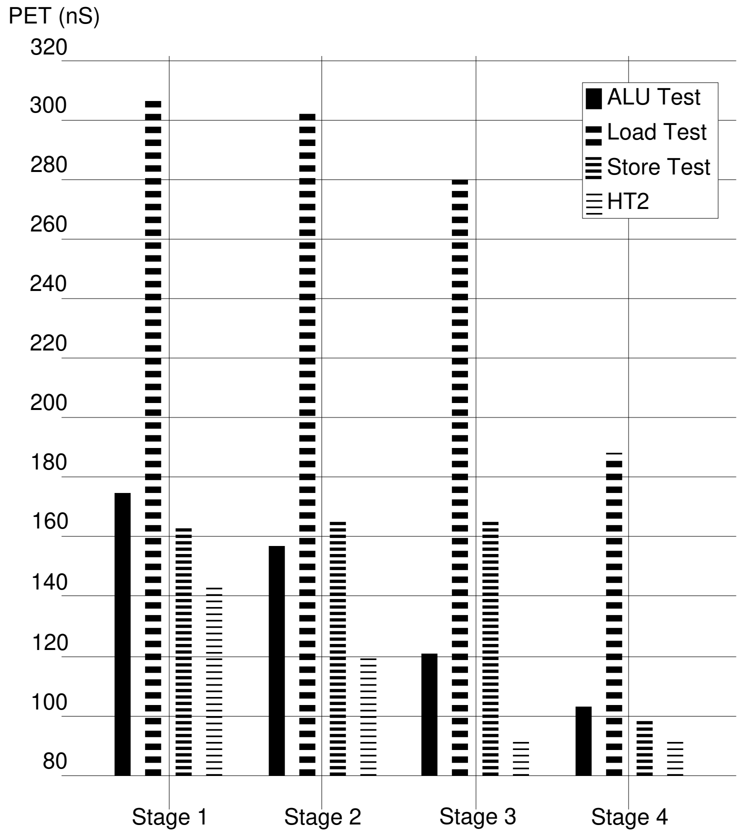


Figure 4: Comparison of Execution Times of the Test Programs

the FM delays for the Store and ALU operations, the scope for spatial concurrency in this particular example is quite small). As the number of microagents in each stage is increased, the spatial concurrency effect will be more pronounced. The speed-up for HT2 as expected reflects the reduced concurrency which can be exploited, due to the data dependencies in the program.

In summary, the rôle of the CU in an asynchronous processor has been considerably simplified to just initiating individual microoperations as early as possible. The control of the datapath is distributed to local interfaces, courtesy of the micronet.

4 CONCLUSIONS

This work has investigated the influence of an asynchronous control paradigm on the design and performance of processor architectures. By viewing the datapath as a network of microagents which communicate asynchronously, one can extract fine-grain concurrency between and within instructions. The micronet can be easily implemented using simple self-timed elements such as Muller C-elements [7] and conventional gates. Future work will investigate the suitability of asynchronous processors as targets for optimising compilers.

Acknowledgements

V. Rebello was supported by the U. K. Engineering and Physical Sciences Research Council (EPSRC) through a postgraduate studentship. This work was partially supported by a grant from EPSRC entitled *Formal Infusion of Communication and Concurrency into Programs and Systems* (Grant Number GR/G55457).

References

[1] D. K. Arvind and V. E. F. Rebello. On the performance evaluation of asynchronous processor architectures. In P. Dowd and E. Gelenbe, editors, *Proceedings of the 3rd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'95)*, pages 100–105, Durham, NC, USA, January 1995. IEEE Computer Society Press.

[2] European Silicon Structures Limited. *Solo 1400 Reference Manual.* ES2 Publications Unit, Bracknell, U.K., 1990.

[3] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In T. Yanagawa and P. A. Ivey, editors, *The Proceedings of the IFIP International Conference on Very Large Scale Integration (VLSI'93)*, pages 5.4.1–5.4.10, Grenoble, France, September 1993.

[4] J. Hennessy and T. Gross. Postpass code optimisation of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[5] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *The Proceedings of ASPLOS III*, pages 272–282. ACM Press, April 1989.

[6] C. Mead and L. Conway. *Introduction to VLSI Systems.* Addison-Wesley, Reading, Mass., 1980.

[7] R. E. Miller. *Switching Theory. Volume II: Sequential Circuits and Machines.* John Wiley and Sons, 1965.

[8] W. F. Richardson and E. L. Brunvand. The NSR processor prototype. Technical Report UUCS-92-029, Department of Computer Science, University of Utah, USA., 1992.

[9] R. F. Sproull, I. E. Sutherland, and C. E. Molnar. Counterflow pipeline processor architecture. Technical Report SMLI TR-94-25, Sun Microsystems Laboratories Inc., April 1994.

[10] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[11] J. E. Thornton. *Design of a Computer: The Control Data 6600.* Scott Foresman and Company, 1970.

[12] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

D.2 On the Performance Evaluation of Asynchronous Processor Architectures

Title:	On the performance evaluation of asynchronous processor architectures.
Authors:	D. K. Arvind and **V. E. F. Rebello**.
Presented at:	The 3rd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'95).
Place:	Durham, NC, USA.
Date:	18^{th} – 20^{th} January 1995.
Publisher:	IEEE Computer Society Press.

On the Performance Evaluation of Asynchronous Processor Architectures

D. K. Arvind and V. E. F. Rebello
Department of Computer Science, The University of Edinburgh
Edinburgh, EH9 3JZ, United Kingdom
E-mail: {dka,vefr}@dcs.ed.ac.uk

Abstract

This paper evaluates and analyses the influence of an asynchronous control paradigm on the performance of processor architectures. The idea of a micronet is introduced which models the datapath as a network of concurrent functional units which communicate with each other asynchronously. This allows the efficient exploitation of fine-grained instruction-level parallelism (ILP). A micronet-based asynchronous processor (MAP) architecture is described in Occam2 and simulated in a parallel discrete event simulation environment. Suitable metrics are introduced for measuring the performance of the MAP datapath.

1 Introduction

There has been renewed interest in asynchronous circuits especially in a restricted form known as self-timed circuits [14]. These circuits have a number of advantages [11], including their automatic synthesis from specifications [7]. While this has resulted in provably-correct circuit designs, the performance of the resulting processor architectures have been largely overlooked [4, 12].

A few processors have been proposed [6, 13] which utilise asynchrony at the circuit level and exploit average-case behaviour for better performance. An examination of the influence of asynchronous control paradigm on the design of processor architectures has recently been reported [1]. A new model has been proposed called the *micronet* for modelling asynchronous datapaths, which efficiently exploits instruction-level parallelism in programs.

The designs in [6, 13] are based on a single micropipeline datapath [15]. Viewing the datapath as a linear sequence of stages may not be very efficient for reasons elaborated in the following section. This paper evaluates the performance of an asynchronous datapath based on the micronet model which treats the datapath as a network of communicating functional units called *microagents*.

2 Asynchronous Pipelines

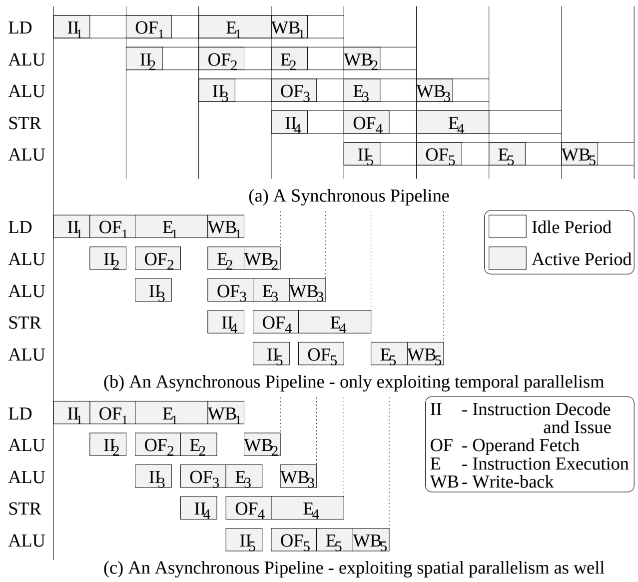


Figure 1: Synchronous and Asynchronous Pipelines

The clock period of a synchronous pipeline is determined by the delay of the slowest stage which takes into account worst-case timings for execution and propagation. Furthermore, optimal performance for a pipeline is achieved when all the stages are balanced. This is quite difficult to achieve in practice, since the stages of a typical pipeline perform different operations, and often their delays are data-dependent. Figure 1(a) illustrates the operation of such a datapath in which synchronisation overheads have been omitted for the sake of brevity. This imbalance between stage delays results in idle periods leading to poor utilisation of the physical resources. Of course, further pipelining of the slower stages could reduce this at the cost of increased design complexity and synchronisation overheads.

In contrast, the performance of an asynchronous pipeline is determined by the actual delays of individual stages (usually the average delays), and over-

Published in the Proceedings of the 3rd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'95), pp 100-105, Durham, NC, USA, January 1995.

heads due to self-timing protocols (which have been omitted in Figure 1(b), but have been included in the models). This pipeline only exploits temporal parallelism as before, but does so more efficiently. We make some further observations about the stages in a synchronous datapath. All the instructions may not require the services of all the stages. Secondly, although each stage may consist of different resources, only one of them will be active at any time for a given instruction. Figure 1(c) illustrates an asynchronous pipeline which exploits spatial parallelism within some of the stages. Successive instructions which utilise different resources within a stage are now able to execute concurrently. In the simple example under consideration in Figure 1(c), the execute stage has two concurrently-operating resources. It is possible for the instructions to share resources in any of the stages. For example, while an instruction is stalled waiting for an operand on one bus, another instruction could use the other buses to fetch its operands. The amount of spatial parallelism which can be exploited in practice is determined by the relative delays of the functional units in the datapath (see Section 4.2 for more details). The next section briefly describes *micronets* which can be used to model asynchronous datapaths.

3 Micronets

Micronets can be viewed as a generalisation of Sutherland's micropipelines [15]. A micronet is described as a network of elastic pipelines in which individual stages of the pipelines have concurrent operations, and stages of different pipelines can communicate with each other asynchronously. The operations of a micronet stage can be exposed as fine-grained *microagents*. This should not be confused with further pipelining of each of the stages. In fact microagents within each stage operate concurrently and can communicate asynchronously with microagents of any of the other stages. A microagent fires when the set of inputs determined by the control signals are valid, and generates a set of outputs. Each program instruction spends time only in the relevant stages and for just as long as is necessary. Furthermore, the different microagents within a stage which belong to different program instructions operate concurrently.

Synchronous datapaths require either software or hardware interlock mechanisms to resolve data dependencies [8], and scoreboards to avoid structural hazards. However, a micronet-based datapath uses existing handshaking mechanisms and register locking to attain the same effect. Out-of-order instruction completions can be easily achieved, thereby further exploiting ILP in the programs. In the following section the performance evaluation of a micronet-based asynchronous processor is presented.

4 Performance Evaluation of MAP

A MAP architecture can be viewed as an ensemble of heterogeneous functional units which operate concurrently and communicate with each other asynchronously. We wish to accurately measure the performance of programs on such an architecture, and to observe the effects of architectural changes. For our purposes the architecture is modelled at the register-transfer level in the Occam2 language [9], with accurate timing delays of the functional units being provided by SPICE-level simulations of their VLSI implementations. Occam2 is based on the process model view of computing in which a system can be described as a collection of concurrent processes which communicate with each other asynchronously through channels. The simulation platform is a transputer-based ME*i*KO Computing Surface [10]. The underlying timekeeping mechanism is based on a parallel asynchronous simulation algorithm [2], which efficiently simulates the class of architectures under investigation.

4.1 The MAP Datapath

The datapath can be described as a network of microagents (denoted by solid boxes) which are connected via ports as illustrated in Figures 2 and 3. The Functional Microagents (FMs) perform microoperations which are typical of a datapath. On each port of a FM is a Communicating Microagent (CM) which is responsible for communications among the FMs, and with the Control Unit (CU). The FMs are effectively isolated and only communicate through their CMs, and can therefore be modified without affecting the rest of the micronet.

The processor design as illustrated in Figure 2 only exploits the actual execution times of microoperations (MAP 1), whereas the design as shown in Figure 3 exploits both this property and concurrency between the microoperations of different instructions (MAP 2). In both cases, each microoperation is initiated by four-phased control signals from the CU, whose acknowledgements are used as status flags for mimicing a scoreboard.

4.1.1 Instruction Issue and Data Transfer

All the microoperations for an instruction are initiated in unison, with the next set waiting until the completion of the previous one. The start of a microoperation is acknowledged which results in the de-assertion of the initiating control signal. The subsequent instruction can only be issued once the previous set of control signals have all been acknowledged which effectively serialises the instruction execution. In MAP 2, the CU initiates the microoperations individually for the

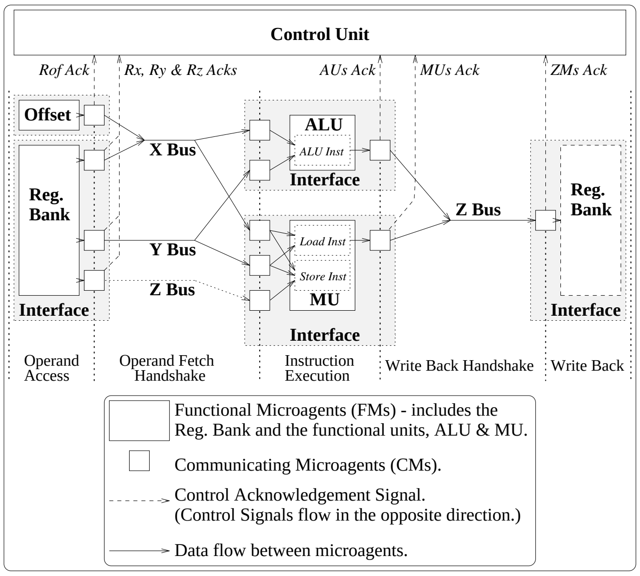


Figure 2: The micronet model of MAP 1

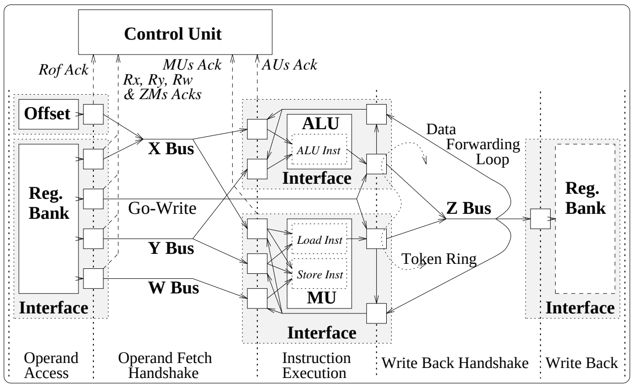


Figure 3: The micronet model of MAP 2

current instruction as early as possible via the corresponding CMs. The receipt of the acknowledgement only confirms that the CMs will initiate the corresponding microoperation. This allows microoperations relating to different instructions to overlap. Hazard avoidance is implicit in the orderings of the assertions of the control signals [1]. The rôle of the CMs has been enhanced to effectively buffer the initiations of the microoperations from the CUs until the respective FMs are ready to perform. The writing back to the register bank is no longer controlled by the CU, but directly by the CMs of the FMs which require the service. These features help to exploit more finer-grained concurrency between instructions than previously possible. In MAP 2, out-of-order instruction completion (due to different execution delays in the FMs) and data-forwarding are also supported [1].

In the next section the effect of these features on the performance of simple programs are investigated by simulating the micronet model in a parallel discrete event simulation environment which was briefly described earlier.

4.2 Performance Results

The performance evaluation of asynchronous pipelines is non-trivial since the stage delays are non-uniform, and variable due to data dependencies. The interaction between successive instructions which leads to spatial and temporal concurrency is difficult to evaluate accurately through analytical methods. The two principal attributes which affect the performance of programs in asynchronous pipelines are the *latency* of the relevant microagents, which is defined as the time between initiating the microoperation and the result being available, and their *cycle time*, which is the minimum time between successive initiations of the same microoperation, i.e. throughput. (They are the same in a synchronous pipeline, with the cycle time being determined by the slowest latency.) The difference between the two values can be viewed as the overhead due to asynchronous protocols and a good design should endeavour to minimise it. This is achieved in micronets by overlapping the phases of the communication protocol in CMs with useful operations in the FMs, thus hiding the overhead. The effectiveness of this method can be determined by measuring the utilisation of FMs by exercising them with test programs composed of appropriate, identical instructions. A few metrics are now introduced for gauging the performance of micronet datapaths.

Minimum Datapath Latency (MDL) - The time between asserting the control signals (i.e. initiating an instruction issue) and receiving the final acknowledgement of the instruction's completion.

Instruction Cycle Time (ICT) - The time between two identical instruction issues once that instruction's pipeline is full. In asynchronous pipelines which usually have non-uniform stage delays, the time between successive instruction issues is influenced by the slowest stage *currently* active in the pipe.

Program Execution Time (PET) - The actual execution time of the program.

ALU Utilisation - The percentage of the program execution time for which the ALU performs useful computation.

MU Utilisation - Same as above, but for the Memory Unit (MU).

Maximum FM Utilisation (MFU) - The upper bound on the FM utilisation is the ratio of the FM's microoperation latency and the ICT.

Inst	MAP 1			MAP 2		
	ICT	MDL	MFU	ICT	MDL	MFU
ALU	24nS	24nS	16.7%	12nS	24nS	33.3%
LD	43nS	43nS	53.5%	23nS	43nS	100%
ST	23nS	21nS	42.9%	12nS	21nS	75%

Table 1: Instruction Execution

Test Pgs	ATP	LTP	STP	HT1 & 2
PET	168nS	301nS	159nS	136nS
ALU Util	16.6%	0%	0%	8.4%
MU Util	0%	53.3%	39.9%	22.4%

Table 2: Execution of Test Programs on MAP 1

The Alu, Load and Store test programs (ATP, LTP, STP) measure the maximum attainable utilisation of their respective FMs. Each contains repetitions of either ALU, LOAD or STORE instructions, such that only structural dependencies exist between instructions (in effect setting up a static pipeline or a fixed path through a network of components). The number of instructions in the test programs are sufficient to fill the pipeline, i.e. enough instructions exist to allow the CU to achieve a steady issue rate. The Hennessy Test (HT1) consists of a mix of the previously-mentioned instructions without any data dependencies, which exercises the spatial concurrency and out-of-order completion, for a particular schedule devised by the compiler. HT2 is a variant of HT1 with data dependencies, which exercises the data forwarding mechanism as well.

The functional units were implemented in a 1.5 μm CMOS process. The timing characteristics were extracted from a post-layout simulation tool within a commercial VLSI design package called SOLO 1400 [5] and incorporated into the Occam2 model.

In MAP 1, the ICT value for each instruction is determined by the slowest microagent control signal handshake required by that instruction, since the next instruction issue cannot begin until all the previous handshakes have been completed. The results in Table 1 show that the ICT is equal to the MDL (except for the ST instruction), which is not surprising as instructions execute sequentially but only take as long as is necessary. The higher value for the ST instruction is due to a handshake delay, which in the case of the LD instruction is hidden by the write-back stage. Although there is no explicit pipelining of the datapath, different phases of the handshaking may occur at the same time, e.g. a CM may initiate a handshake with another CM while completing one with its FM.

Test Pgs	ATP	LTP	STP	HT1	HT2
PET	96nS	181nS	93nS	72nS	84nS
Spd Up	1.75	1.66	1.71	1.89	1.62
A Util	28.9%	0%	0%	16.4%	14.1%
M Util	0%	88.5%	67.7%	43.8%	37.7%

Table 3: Execution of Test Programs on MAP 2

Also in Table 1, the maximum FM utilisations represents the proportion of the MDL taken by the FM to complete its operation. As expected, the execution times of the test programs in Table 2 are the sum of their individual instruction execution times. We observe that the utilisations achieved for the FMs (in Table 2) are very close to their upper bounds (in Table 1) which shows that asynchronous control using a micronet can be efficient.

The ICT figure for the LD instruction in MAP 2 is the best attainable as it represents the MU delay for the operation. The corresponding utilisation figure in Table 3 supports this claim (Note: these utilisation measurements do not take into account both the initial operand fetch and the final write-back delays, and will therefore never attain the theoretical upper bound). These figures show that the micronet can exploit the actual operational costs and effectively hide the overheads of self-timed design. The ICTs for the ALU and ST instructions are limited by their operand fetch cycle times, and the utilisation of the FM in these cases also approach their bounds. This cycle time is due to the communication protocol between the FUs and the register bank. These delays can be reduced by using a less conservative bundling delay [15] and through layout and transistor size optimisation [3].

The improvements in the program execution times (PET) for MAP 2 (shown in Table 3) for the three instruction test programs are due to improvements in temporal concurrency due to asynchronous pipelining of the datapath. Although the actual speedups achieved are less than the ratios of the ICTs for MAP 1 and MAP 2 (shown in Table 1), they are the maximum attainable improvement. The speed-up for HT1 is in part due to the pipelining of the instructions as observed previously in the other test programs, and also due to additional spatial concurrency through overlapping of different instructions in the same stage of the micronet. This further improvement is still significant (approximately 17% in this example) given that successive instruction operand fetches and write-backs are effectively forced to take place sequentially due to resource constraints. (In fact, since these delays are larger than the FM delays for the Store and ALU operations, the scope for spatial concurrency in this particular example is quite small.) As the number of microagents in each stage is increased, the spatial concurrency effect will be more pronounced, subject to relative delays of the microagents. The speed-up for HT2 as expected reflects the reduced concurrency which can be exploited, because of data dependencies in the program.

It has to be noted that the datapath latency is unaffected by the exploitation of temporal parallelism which is generally not the case in a synchronous pipeline.

The interaction between concurrently executing instruction is quite difficult to predict. For example, two instruction which compete for the same resources might acquire them in different order depending on the actual delays which are themselves data-dependent. This is not in itself a drawback, since one of the instruction is stalled for just as long as is necessary, which would not be true in a synchronous case.

5 Conclusions

The behaviour of asynchronous processors are complex and their performance is difficult to predict. Discrete event simulations as described in this work offer a method for accurately measuring their performance. The model in Occam2 naturally captures the concurrency and asynchronous communication. This also allows the simulation to be parallelised to obtain reasonable run-times for large circuits and test programs. This is aided by the asynchronous nature of the underlying simulation algorithm itself.

To the best of our knowledge this is the first work which has investigated the influence of an asynchronous control paradigm on the performance of processor architectures for exploiting fine-grained ILP. The micronet model allows the exploitation of both temporal and spatial concurrency which results in efficient utilisation of resources within the datapath.

Acknowledgements

V. Rebello was supported by a postgraduate studentship from the U. K. Engineering and Physical Sciences Research Council (EPSRC). This work was partially supported by a grant from EPSRC entitled *Formal Infusion of Communication and Concurrency into Programs and Systems* (Grant Number GR/G55457).

References

[1] D. K. Arvind and V. E. F. Rebello. Instruction-level parallelism in asynchronous processor architectures. In M. Moonen and F. Catthoor, editors, *Proceedings of the 3rd International Workshop on Algorithms and Parallel VLSI Architectures*, pages 203–215, Leuven, Belgium, August 1994. Elsevier Science Publishers.

[2] D. K. Arvind and C. R. Smart. Hierarchical parallel discrete event simulation in composite ELSA. In *Proceedings of the Sixth Workshop on Parallel and Distributed Simulation (PADS'92)*, pages 147–156, January 1992.

[3] S. M. Burns. *Performance Analysis and Optimisation of Asynchronous Circuits*. PhD thesis, Computer Science Department, California Institute of Technology, Pasadena, California, USA, 1991.

[4] I. David, R. Ginosar, and M. Yoeli. Self-timed architecture of a reduced instruction set computer. In S. Furber and M. Edwards, editors, *The Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies*, Manchester, UK, March 1993. Elsevier Science Publishers.

[5] European Silicon Structures Limited. *Solo 1400 Reference Manual*. ES2 Publications Unit, Bracknell, U.K., 1990.

[6] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In T. Yanagawa and P. A. Ivey, editors, *The Proceedings of the IFIP International Conference on Very Large Scale Integration (VLSI'93)*, pages 5.4.1–5.4.10, Grenoble, France, September 1993.

[7] S. Hauck. Asynchronous design methodologies: An overview. Technical Report TR 93-05-07, Department of Computer Science and Engineering, University of Washington, Seattle, USA, 1993.

[8] J. Hennessy and T. Gross. Postpass code optimisation of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[9] INMOS Limited. *Occam2 Reference Manual*. Prentice Hall International, 1988.

[10] INMOS Limited. *Transputer Reference Manual*. Prentice Hall International, 1988.

[11] A. J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. Technical Report Caltech-CR-TR-89-1, Department of Computer Science, California Institute of Technology, Pasadena, California, 1989.

[12] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus. The design of an asynchronous microprocessor. In C. L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 351–373, Cambridge, Mass., 1989. MIT Press.

[13] W. F. Richardson and E. L. Brunvand. The NSR processor prototype. Technical Report UUCS-92-029, Department of Computer Science, University of Utah, USA., 1992.

[14] C. L. Seitz. System Timing. In C. Mead and L. Conway, editors, *Introduction to VLSI Systems*, chapter 7, pages 218–262. Addison-Wesley, 1980.

[15] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

D.3 A Model for Decentralising Control in Asynchronous Processor Architectures

Title:	Micronets: A model for decentralising control in asynchronous processor architectures.
Authors:	D. K. Arvind, R. D. Mullins and **V. E. F. Rebello**.
Presented at:	The 2nd Working Conference on Asynchronous Design Methodologies.
Place:	London, UK.
Date:	30^{th} – 31^{st} May 1995.
Publisher:	IEEE Computer Society Press.

Micronets: A Model for Decentralising Control in Asynchronous Processor Architectures

D. K. Arvind, R. D. Mullins and V. E. F. Rebello
Department of Computer Science, The University of Edinburgh
Edinburgh, EH9 3JZ, United Kingdom
E-mail: dka@dcs.ed.ac.uk

Abstract

Micronets model processor architectures as a network of communicating resources, in contrast to the traditional one of a linear pipeline. Micronets distribute the control to the functional units, which enables the exploitation of fine-grain concurrency between instructions. The overhead due to asynchrony is hidden with the four-phase protocol being used to implement scoreboarding and hazard avoidance mechanisms, without incurring additional control costs. This paper demonstrates the feasibility of micronet-based processors. Results are presented for SPICE-level simulations of a 0.7μm CMOS implementation of a datapath. The relationships between micronets and both the compiler and the computer architecture are also explored.

1 Introduction

Micropipelines [22] have been used to model linear asynchronous pipelines such as datapaths [6] [18], and two-dimensional pipeline structures [8]. However, viewing a datapath as a single linear pipeline has limitations [2]. A new paradigm called micronets has recently been proposed for the distribution of control in asynchronous processor architectures [1]. Micronets model datapaths as a network of communicating functional units which allows the efficient exploitation of both fine-grained instruction-level parallelism and the actual execution costs of instructions.

The choice of a four-phase communication protocol [19] between the functional units allows the efficient utilisation of these resources, by avoiding the additional control costs (scoreboarding and hazard avoidance mechanisms) normally associated with processors which exploit ILP.

The design of an effective micronet-based system should also consider the interplay between the compiler and the processor architecture, i.e. does a micronet-based processor offer a good target for a parallelising compiler. The influence of this asynchronous target on compiler design is briefly discussed.

2 Micronets

Micronets are a generalisation of micropipelines. The operations within each of the micropipeline stages are exposed in the form of fine-grain *microagents*. The microagents in any "stage" can operate concurrently, and microagents in the different "stages" communicate with each other asynchronously. Program instructions only utilise the relevant microagents and for just as long as is necessary. More than one instruction may utilise different microagents within a "stage". Figure 1 compares the resource utilisation in micropipelined and micronet datapaths. In the former, the number of active instructions is never greater than the number of pipeline stages, and at any time only a subset of the resources in each of the stages is normally utilised. In micronets the number of instructions which may be active at any time is bounded by the number of microagents. An instruction which does not require any of the resources within a "stage" can skip it. Furthermore, the time spent by instructions in microagents may vary. Due to these reasons instructions may overtake. Synchronous processors which permit this do so at a significant control cost; the resulting scoreboarding mechanism is also used to resolve structural hazards (together with software or hardware interlocks to resolve data dependencies). All of this comes for free in a micronet-based datapath: the existing handshaking mechanisms and register locking provide these services [1].

In practice datapaths have to deal with conditions which interrupt the flow of instructions, such as condition branching and exception handling. The RISC architectures have popularised the "delayed branching" approach to reducing the performance degrada-

Published in the Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies, pp 190-199, London, UK, May 1995.

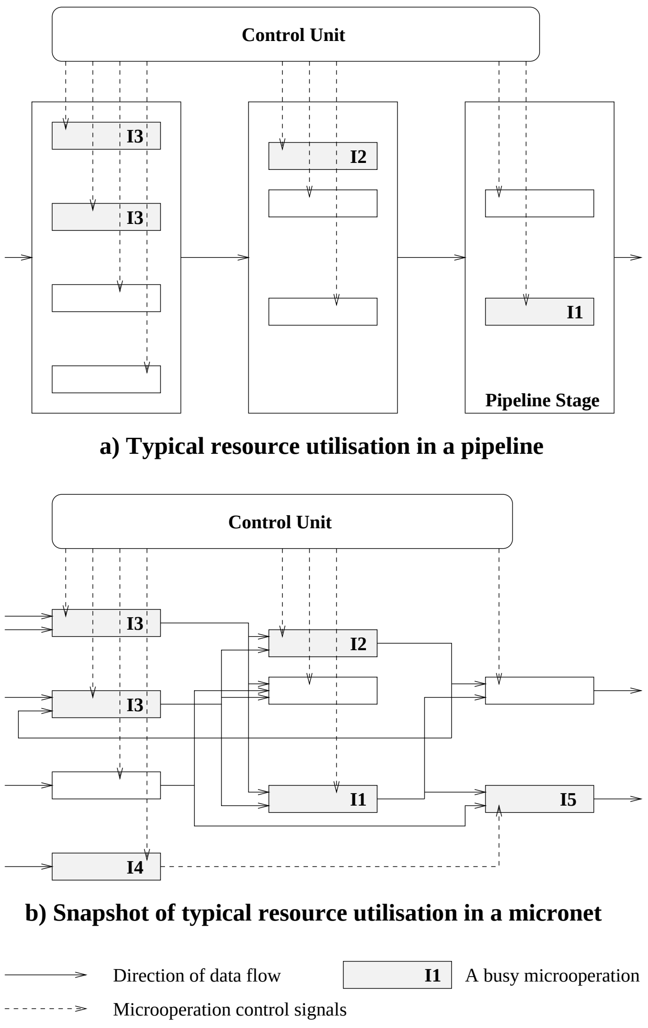


Figure 1: Contrasting a micropipeline with a micronet

tion of condition branching [9] [16] [17]. However, this technique is unsuitable for asynchronous datapaths because of the difficulty in estimating the time to resolve the branch condition (which is fixed in synchronous architectures). Therefore, the number of instructions which have to be fetched cannot be determined. For micronet-based architectures, the preferred techniques are ones which do not rely on fixed timing for their correct operation, such as branch prediction schemes [12] [20] or advanced branching mechanisms [15]. Precise exception handling and speculative execution are supported through the use of history and write-back buffers [21] [23].

A micronet-based datapath, as illustrated in Figure 2, is composed of a network of microagents (denoted by solid boxes) which are connected via ports. The Functional Microagents (FMs) perform microoperations which are typical of a datapath. On each port of a FM is a Communicating Microagent (CM) which is responsible for communications among the FMs, and with the Control Unit (CU). The FMs are effectively isolated and only communicate through their CMs, and can therefore be modified without affecting the rest of the micronet. The protocol used in the design of micronet-based datapath is discussed in the following section.

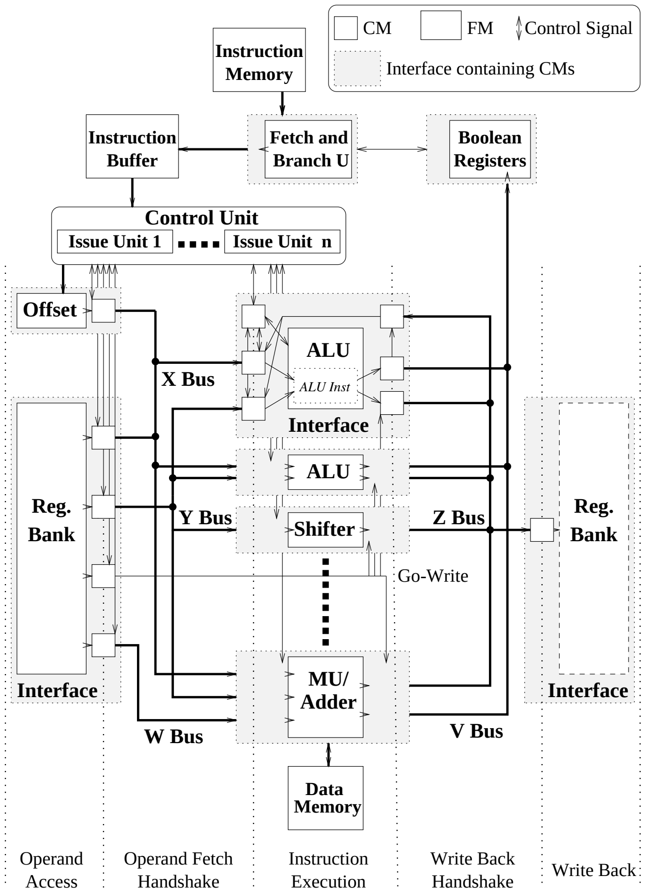


Figure 2: A micronet-based processor architecture

2.1 Choice of protocol

Both transitions in a generic four-phase protocol (the assertion and the return-to-zero) are accompanied by additional acknowledgements from the receiver. The principal advantage of this approach is a simpler circuit implementation. However, it uses twice as many transitions than is necessary and whenever the wire delay is a substantial fraction of the operation time, the extra trip required by a single communication can be a serious performance penalty. In fact, the reset phase of the handshake does not signal any event, thus leading some designers to modify the protocol to simultaneously reset the two signals after the active phase to reduce the handshake cycle time [5]. The micronet is only concerned with the external communications between microagents, which might use a different protocol internally. Micronets employ the traditional four-phase handshaking protocol for both control and local bundled data transfer. Other reasons, more specific to micronets, have influenced this choice, and these are discussed next.

2.1.1 Fast instruction issue

One of the significant features of micronets is its ability to exploit spatial concurrency within the datapath. This requires a fast instruction issue rate to keep the microagents busy. The CU initiates the microoperations for each of the instructions individually and as early as possible. The acknowledgements from the CMs (after a delay of one C-element) confirm that the corresponding microoperations will be initiated. The instruction is considered to have been issued once the CU has received all the acknowledgements. This corresponds to the first half of the four-phase protocol. The CU is free to issue the next instruction, while the reset phase of the protocol completes. This is done when the corresponding acknowledgement signal is deasserted which signifies that the particular resource is ready for the next request. The instruction releases the resources individually as soon as the respective microoperations have completed, freeing the resources for another instruction. Figure 3 shows the activity of two resources in micronets in comparison to a similar synchronous pipeline and micropipeline.

The microoperations of different instructions may overlap leading to potential hazards. Since the acknowledgement signals denote the busyness of resources, they can be collectively used as a scoreboard. Hazard avoidance due to data dependences is implicit in the orderings of the assertions of the control signals [1]. These *pre-issue conditions* stall the assertion of the respective control signal until the completion of one of the halves of the handshake protocol of the dependent microoperation control signal(s).

Although a four-phase protocol would be considered twice as expensive as a two-phase one, the same efficiency is obtained as two back-to-back, two-phase handshakes by representing two events in each cycle. The recovery transitions are used by the control unit for scoreboarding and hazard avoidance. This is necessary for efficient exploitation of ILP, since the control unit has to issue each instruction before the

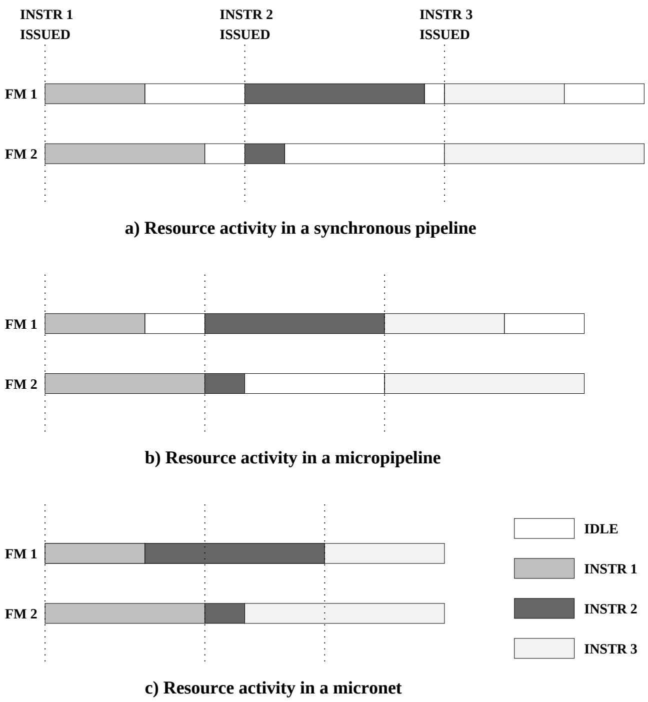


Figure 3: Resource activity

previous one completes its execution. Furthermore, a four-phase protocol exposes more concurrency by effectively decoupling the sender's and receiver's operations from their communication [1].

2.1.2 Routing data in micronets

Although the actual data transfer between microagents is controlled locally via handshake protocols, the access to shared resources, such as data highways, may be controlled either globally by the CU or locally by an arbitration scheme. Global control is used in cases where the order of granting resources is known in advance and has to be enforced. This is again achieved through the use of pre-issue conditions [1]. Otherwise, a local mutual exclusion scheme such as in token rings or arbiters will grant requests. For example, the writing back to the register bank is controlled directly by the CMs of the FMs which require this service. As a consequence of this and also due to the differences in the execution times of microoperations, instructions may complete out of order. Therefore data has to be tagged with its destination which also enables data-forwarding to be supported.

The reader is referred to [1] for further information on micronets, and to [2] for the performance evaluation of micronet-based datapaths.

3 Implementing a micronet-based datapath

The Control Unit (CU) is essentially an instruction queue which is now reduced to simply issuing instructions. The CU makes requests to the interfaces of the functional units, with the instruction's completion being controlled by these interfaces, thereby leaving the issue unit free to rapidly initiate the next instruction. The microoperations associated with these instructions operate concurrently, subject to dependence rules.

The destination registers can be locked to ensure that they are accessed in the correct order. In addition, a number of pre-issue constraints or dependence rules are implemented in the control unit to prevent Write-after-Read (WAR) and Read-after-Write (RAW) pipeline hazards. The rules may be relaxed by checking for dependencies between specific registers which are required by the current instruction. A Go-write mechanism maintains the correct ordering of write-backs to the same register, thus avoiding Write-after-Write (WAW) errors, without stalling the issue of instructions [1].

Figure 4 illustrates the instruction issue logic. Only a selection of microagents are required on each instruction issue cycle. A dual-rail encoder is used to produce both the requests for the necessary microagents and a complemented signal to mask those which are not required. Microagents which are not involved in the current instruction issue will not block due to this masking mechanism.

The dependence rules, P_0 to P_n, represent the specific conditions that must be met before that microoperation is issued. Each wait operation is implemented as a single asymmetric C-element, which ensures that any subsequent clearing of the pre-issue constraint does not result in another issue request.

The ALU and its interfaces are shown in Figure 5. The interface or control logic for the ALU can be decomposed into two main operations - execution, and write-back, which operate concurrently. Execution involves the fetching of operands, the go-write mechanism and the ALU operation itself, while the write-back allows the result to be written to the destination register. On the issue of an ALU microoperation, execution progresses as follows:

- The acknowledge to the control unit is asserted to signal that the microoperation has begun.

- The execute interface initiates the go-write and both operand requests.

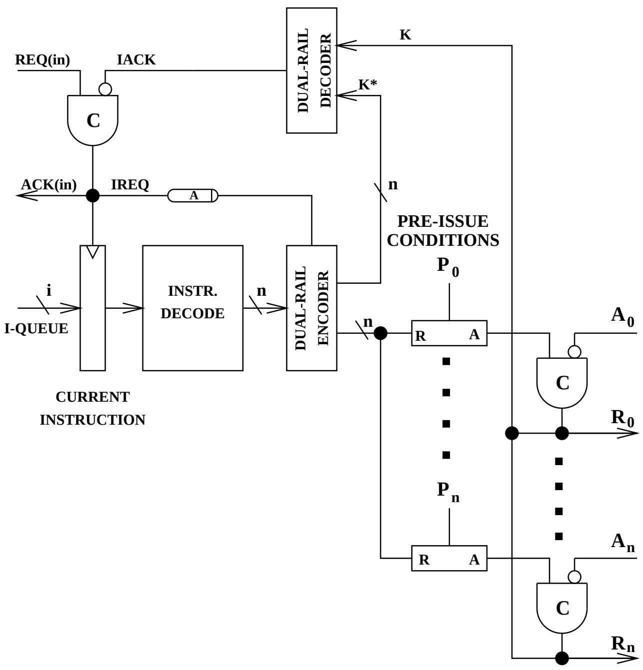


Figure 4: Instruction issue logic

- The operand interfaces make requests to the register bank for their operands (operand register details are provided directly by the control unit). A go-write request is also made by the go-write interface.

- The ALU's operand fetch interfaces signal the receipt of data to the ALU's execute interface, and complete their handshakes with the register interface.

- Operands are dual-rail encoded to allow a completion signal to be generated.

- The ALU operation begins, the result is detected and the execute interface attempts to issue a write back operation to the ALU's Z-bus interface.

- If the Z-bus interface is free then the result is latched and the ALU acknowledge to the control unit is deasserted. The control unit may now issue the next ALU microoperation.

- If a go-write permission has been received, then the write back occurs as soon as it obtains access to the Z-bus.

A closer inspection of the behaviour of the interfaces shows that the overhead in using a four-phase handshaking protocol is indeed hidden, as shown in the example that follows for the operand interface.

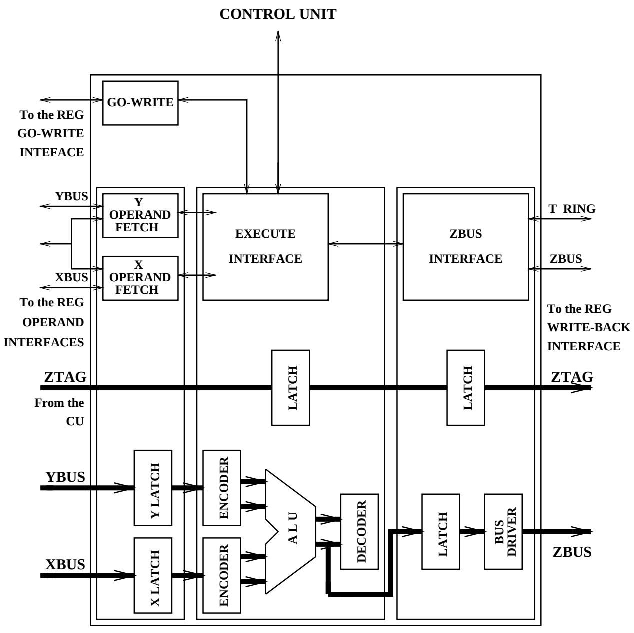


Figure 5: Interfaced ALU

Phase 1 - Request is made to the register bank for an operand. This phase is usually hidden since it takes place concurrently with the operand's register access.

Phase 2 - Acknowledge or data valid signal is received, the receipt of operands is now detected by the control unit and the ALU operation may begin.

Phases 3 and 4 - Handshake completes concurrently with the ALU operation, and these extra phases are effectively hidden.

The destination register for each ALU operation is stored in two tag latches, as shown in Figure 5. The tag and data are sent together to the register block allowing the correct destination register to be selected.

The functions of the register interfaces as shown in Figure 6 are listed below:

Operand Interfaces - These interfaces communicate with the control unit, the register bank and the operand fetch interfaces of other functional units, to control the supply of operands. An operand may only be sent to a functional unit when the following operations have been successfully completed:

- A request has been made to the operand interface by the CU.
- The register's lock bit has been read as clear.
- A request has been made by the functional unit for the operand.

Lock Interface - The behaviour of the lock interface is similar to that of the operand interfaces:

- A request is made by the CU to lock the destination register, which is locked when ready.
- The Go-Write permission is then granted to the requesting functional unit.

Write-Back Interface - This interface accepts data and register tags from the Z-bus and writes the data back to the appropriate register. The corresponding lock bit is then cleared.

Accesses to both the data registers and the lock bits is controlled by using a fixed delay. As the register access time is almost constant, there are few advantages to be gained by implementing a "data valid" signal.

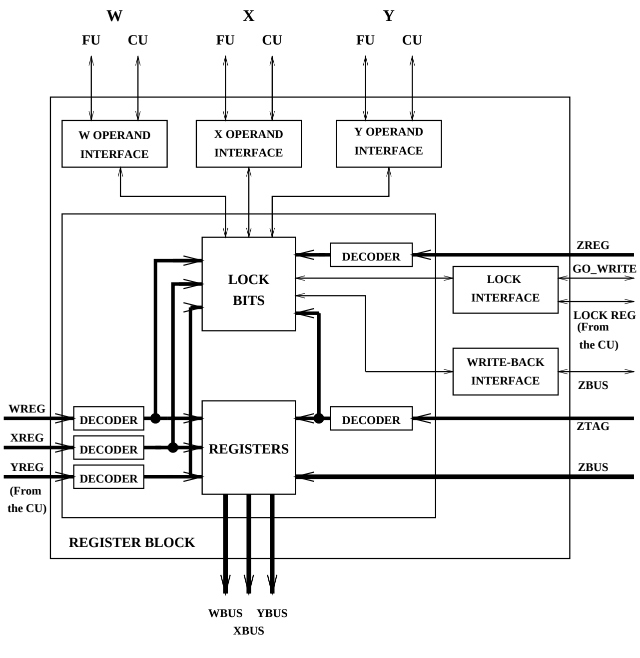


Figure 6: Interfaced Register Bank

Since the write-back bus is shared by a number of functional units, some form of arbitration mechanism must be used to avoid contention, like a token ring. Although easy to implement, its performance would degrade with an increase in resources sharing the bus.

4 Simulation Results

A prototype datapath was implemented in ES2's 0.7μm CMOS process using the Cadence design tools. They were used to create a library of self-timed components and datapath elements. The Cadence Design Framework provided interfaces to both VHDL and HSPICE. A VHDL model of the datapath was created from a high-level specification and synthesised. The HSPICE simulations of the entire datapath took approximately 17 hours on a SUN Sparc-10.

Figure 7 shows the execution of an ALU instruction with traces of the relevant control signals being numbered.

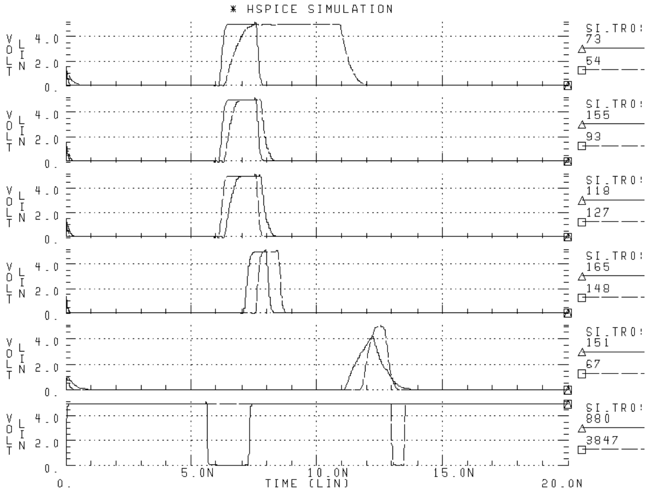


Figure 7: ALU Instruction

Panel 1 - An asserted request signal (**73**) from the control unit to the ALU initiates an add operation. The acknowledge signal (**54**) represents the period of ALU activity.

Panel 2 - The operand request signal (**155**) is sent by the control unit to initiate a register access microoperation. The acknowledge signal (**93**) is asserted by the register operand interface to prevent further operand requests until a functional unit has claimed the current operands. Only the signals for one of the operands is shown.

Panel 3 - A request to lock the destination register (**118**) and the register bank acknowledge signal (**127**) are shown. The acknowledge signal is lowered after the register has been locked and a go-write request has been received from a functional unit.

Panel 4 - Shows an ALU operand request (**165**) to the register bank, together with the corresponding data valid signal (**148**) from the register bank.

Panel 5 - After the add operation has completed, a request (**151**) is made to write the result to the destination register. The receipt of data at the register bank is signaled by the assertion of the acknowledge flag (**67**).

Panel 6 - Shows the instruction decode start signal (**880**). The duration of this signal indicates the instruction issue time. Also shown is the register write signal (**3847**), where data is written back on the final edge of this signal. The complete instruction execution time is represented by the delay between the first and last edges as shown in this panel. (Note that both signals are active low.)

The following subsections describe a number of measurements which influence the performance of micronet-based datapaths.

4.1 Handshaking

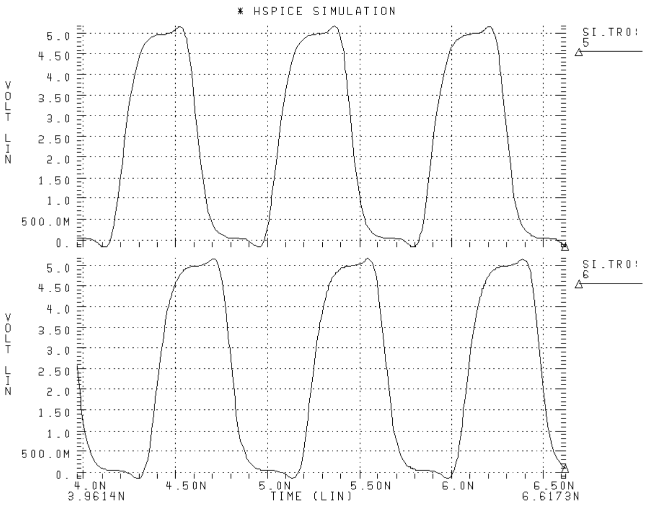


Figure 8: The handshake cycle

The handshake cycle is implemented by two back-to-back C-elements. This forms the basis for distributed control in micronets (a circuit commonly used for communication between microagents) and therefore a crucial factor which influences performance within the micronet. Ignoring the computation within a microagent, the throughput would then be limited by the control handshake cycle. Figure 8 shows a cycle time of 0.8nS, corresponding to a maximum throughput rate of 1.25GHz. This suggests that micronet control circuitry is unlikely to limit throughput in processing pipelines.

4.2 Maximum instruction issue rate

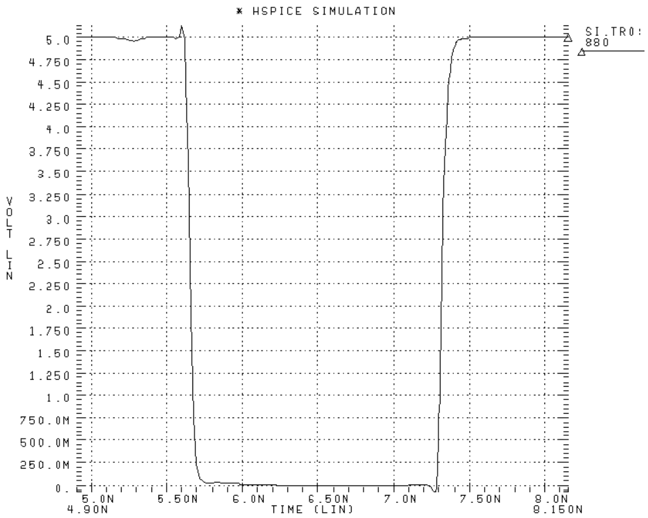


Figure 9: Maximum instruction issue rate

Figure 9 represents an instruction issue time of 1.85nS. The maximum instruction issue rate is determined by the earliest possible reassertion of the issue signal. Given sufficient instruction fetch bandwidth, the minimum cycle time for this signal is 2.05nS which equates to a maximum instruction issue rate of 488Mhz. This represents a theoretical upper limit on processor performance while ignoring datapath delays.

4.3 ALU throughput

Figure 10 shows the signal from the control unit (**73**) being asserted to initiate an ALU operation. The period when both the ALU and its interface are busy is represented by the duration of signal **54** (4.31nS). During this period the ALU interface requests both operands, initiates the operation, detects the result, obtains write-back (go-write) permission and writes the result to the Z-bus. The actual instruction execution time of the ALU is determined by the period between the operands arriving and the ALU's acknowledge being deasserted (3.11nS). This is the delay required to add without any carry propagation and thus represents the minimum time through the functional unit. The minimum ALU instruction cycle time is determined by the earliest possible reassertion of signal **73**. This cycle time was estimated at 4.51nS, implying a peak processor performance of 222 MIPS for add instructions.

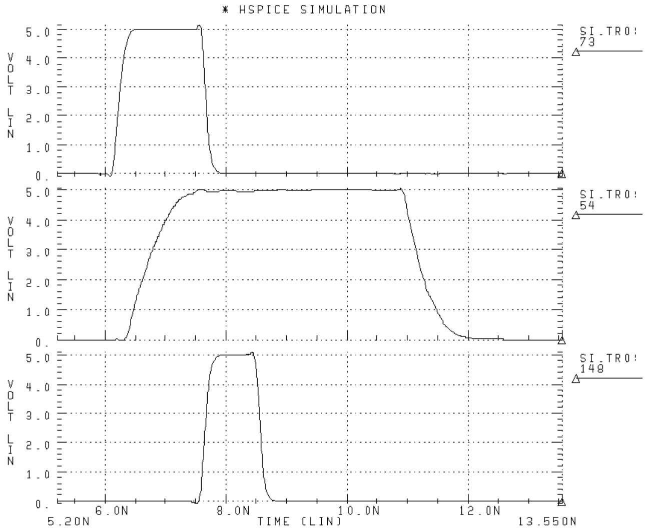


Figure 10: ALU Activity

Only the FM latency should be considered as time spent in useful work, with the other delays being overheads of control paradigm and the architecture. In this implementation, the micronet overhead for this operation is 1.4nS (the difference between the operation's cycle time and its latency). This overhead can be effectively removed by modifying the ALU interface to deassert the microoperation acknowledge signal to the CU once the operands have been fetched [2].

4.4 Operand fetch

The operand fetch delay, as shown in Figure 11, was calculated as the period between the assertion of the operand request signal (**155**) and the assertion of the data valid signal (**148**) from the register bank (1.45nS). The actual time to access one of the registers is determined by the duration between the assertion of the operand request acknowledge (**93**) and the data valid signal (1.24nS).

4.5 Write-back

Figure 12 shows that the time, between the result becoming available at the output of the ALU and being written back in the destination register, is 2.48nS. The actual time taken to write data to a register is 0.5nS (duration of signal **3847**). The slow rising and falling edges of the write-back request (**151**) signal, limits the write-back rate to 474MHz.

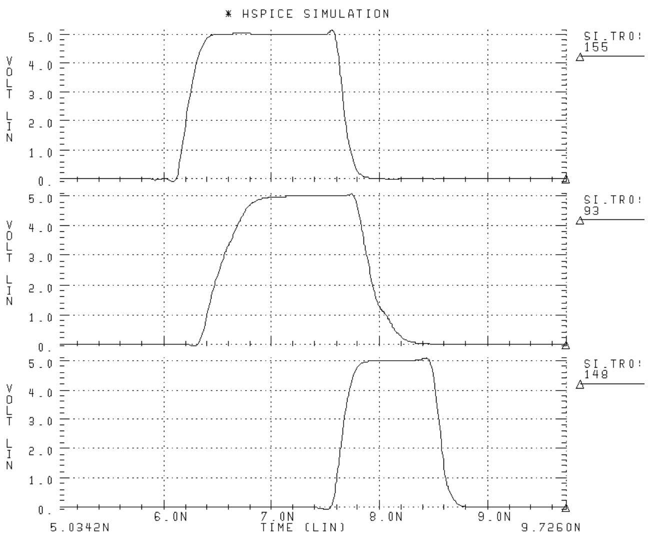


Figure 11: Operand Fetch

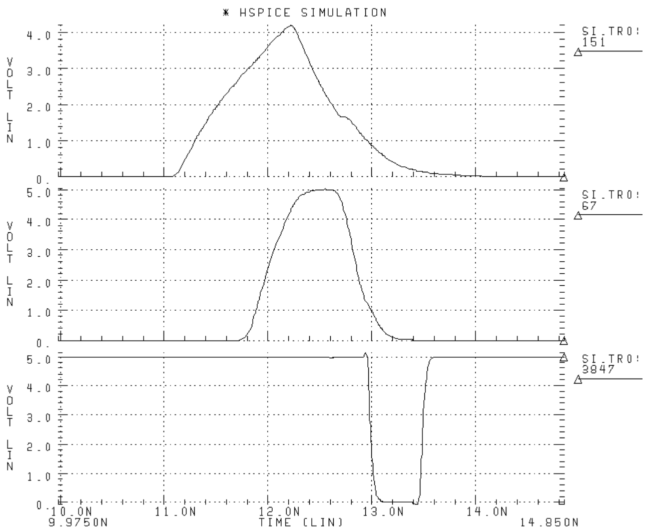


Figure 12: Register Write Back

Note that no circuit optimisation of transistor sizes have yet been made to either improve performance or sharpen edges. Micronet datapaths can be synthesised from high-level specifications using a custom-built library of four-phased self-timed components and interconnection cells.

5 Discussion

The emergence of VLSI technology, together with the maturing of optimising compiler techniques, had aided the development of early RISC architectures [9] [11] [16]. Their primary concern was the efficient usage of expensive silicon real estate, and careful consideration was given to the design of the instruction set architecture [13]. There have been two orthogonal trends in the evolution of synchronous processor architectures [10]: the deeply-pipelined architectures [14], i.e. ones which exploit temporal parallelism, and superscalar architectures which exploit spatial parallelism [3] ([4] is an example which exploits both). Both these classes have benefited from improvements in technology and the resulting faster clock frequencies. But these improvements have been sustained at a high price in terms of clock distribution, power consumption, and design complexity [4]. Furthermore, significant additional control costs are incurred in exploiting ILP in both cases.

Micronets offer an alternative model for the design of future processor architectures. Whereas the original RISC ideal was the efficient usage of the silicon space by identifying the critical resources, we are essentially concerned with their efficient utilisation over time. We achieve this in two ways: by removing the clock, and distributing control to the resources; and viewing the datapath not as a linear pipeline, but as a network of communicating resources. We are able to efficiently (the overheads due to asynchrony are hidden [1]) exploit a fine-grain ILP without the additional control costs (the protocol also implements a scoreboarding and hazard avoidance mechanisms).

The asynchronous and distributed nature of the control in micronets allows the processor to be easily extended with little effect on the rest of the design. For a given class of problems, the designer is able to easily explore the architectural design space more accurately by adding critical resources. This can be naturally extended to superscalar architectures by increasing the number of issue units. (Synchronous superscalar architectures replicate entire datapaths.) The same scoreboarding mechanism is shared between the issue units for determining the global state of the datapath. A micronet-based superscalar architecture has been designed and its performance is currently being evaluated.

5.1 The Micronet and the compiler

The micronet model exposes structural concurrency in the datapath, with fine-grained resources now being visible to the compiler. It is the task of the compiler to schedule instructions such that these resources are efficiently utilised. The instruction schedule is devised based on a model of the architecture; for synchronous architectures the model is simple: instructions do not interact and their execution times are fixed. In contrast, an asynchronous model is necessarily less accurate for the following reason: execution times for the same instruction may vary due to environmental parameters, data-dependent operations, and interactions between different instructions which are simultaneously executing in the micronet. We have considered models based on worst-case instruction execution times where the resulting schedule is treated as a first pass one. The instructions are dynamically reordered at run-time to tune this schedule and due to the asynchronous behaviour these instructions are issued as soon as possible, without the need for delays using NO-OP instructions.

A micronet-based datapath has several communicating "pipelines" which can all potentially be busy simultaneously. The control unit aims to issue those instructions successively which minimise resource contention. It will only stall if no instructions are available for issue, or all the instruction's resources are busy. The micronet's asynchronous behaviour minimises the duration of this stall. In the case of instructions with data or structural hazards, both instructions are issued without stalling, with the second instruction executing until the busy microagent. These fine-grain hazard avoidances are enforced at run-time by the pre-issue conditions of the micronet.

Other reasons contribute towards the complexity of compiler-time scheduling on micronets. An initial state of activity is assumed for scheduling a basic block within the micronet, which might well be different at run-time. The actual state can indeed be determined at run-time thanks to the implicit scoreboarding mechanism in the CU. This information is used to dynamically alter the static schedule by identifying an instruction which can be executed immediately (easily achieved using the control acknowledgement signal), after checking for independence from the previous instructions in the buffer, which is determined at compile-time and marked by a concurrency bit. The instruction issue is only limited by the availability of resources and operands, in the presence of out-of-order instruction issue. Micronets can therefore be viewed as a hybrid dataflow style of architecture which is limited to the window of instructions available in the instruction buffer, without the bookkeeping costs of traditional dataflow architectures [7].

6 Conclusions

We have presented a new model, called micronets, for decentralising controls in asynchronous processor architectures. They are viewed as a network of communicating functional units, which expose fine-grain concurrency between instructions. We have demonstrated that four-phase handshaking protocols enable the implementation of highly concurrent structures and in most cases the overheads can be hidden. Just as importantly, these protocols are used to efficiently avoid datapath hazards.

The modular nature of micronets eases modification and empowers the computer architect with finer control in the design, for example, of superscalar architectures. Some of the issues relating to micronets as targets for parallelising compilers have been discussed.

The control interfaces for the micronet-based datapaths are specified using a library of interconnection cells, and automatically synthesised in terms of simple C-elements. Results from SPICE simulations for an add ALU operation have been presented which demonstrates the feasibility of distributing controls.

In conclusion, the micronet model considers the interactions between the underlying implementation technology, the architecture and the compiler, and underlines our integrated approach to system design.

Acknowledgements

V. Rebello and R. Mullins were supported by postgraduate studentships from the U. K. Engineering and Physical Sciences Research Council (EPSRC). This work was partially supported by a grant from EPSRC entitled *Formal Infusion of Communication and Concurrency into Programs and Systems* (Grant Number GR/G55457).

References

[1] D. K. Arvind and V. E. F. Rebello. Instruction-level parallelism in asynchronous processor architectures. In M. Moonen and F. Catthoor, editors, *Proceedings of the 3rd International Workshop on Algorithms and Parallel VLSI Architectures*, pages 203–215, Leuven, Belgium, August 1994. Elsevier Science Publishers.

[2] D. K. Arvind and V. E. F. Rebello. On the performance evaluation of asynchronous processor architectures. In P. Dowd and E. Gelenbe, editors, *Proceedings of the 3rd International*

Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'95), pages 100–105, Durham, NC, USA, January 1995. IEEE Computer Society Press.

[3] K. Diefendorff and M. Allen. Organisation of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, 12(2):40–63, April 1992.

[4] D. W. Dobberpuhl *et al.* A 200-MHz 64-bit dual issue CMOS processor. *IEEE Journal of Solid-State Circuits*, 27(11):1555–1567, November 1992.

[5] S. B. Furber. Lessons from AMULET1: Towards AMULET2. In *Computing Without Clocks: Asynchronous Microprocessor Design*. The Sun Annual Lecture in Computer Science at the University of Manchester, September 1994.

[6] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In T. Yanagawa and P. A. Ivey, editors, *The Proceedings of the IFIP International Conference on Very Large Scale Integration (VLSI'93)*, pages 5.4.1–5.4.10, Grenoble, France, September 1993.

[7] J.-L. Gaudiot and L. Bic. *Advanced Topics in Dataflow Computing*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1991.

[8] G. Gopalakrishnan. Some unusual micropipeline circuits. Technical Report UUCS-93-015, Department of Computer Science, University of Utah, Salt Lake City, UT, USA, December 1993.

[9] J. Hennessy, N. Jouppi, F. Baskett, and J. Gill. MIPS: A VLSI processor architecture. In *The Proceedings of the CMU Conference on VLSI Systems and Computations*, Rockville, Md. USA., October 1981. Computer Science Press.

[10] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *The Proceedings of ASPLOS III*, pages 272–282. ACM Press, April 1989.

[11] M. G. Katevenis, R. W. Sherbourne, D. A. Patterson, and C. H. Séquin. The RISC II microarchitecture. In F. Anceau and E. J. Aas, editors, *The Proceedings of VLSI'83: VLSI Design of Digital Systems*, pages 349–359. North-Holland, 1983.

[12] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17(1):6–22, January 1984.

[13] Å. Lunde. Empirical evaluation of some features of instruction set processor architectures. *Communications of the ACM*, 20(3):143–153, March 1977.

[14] S. Mirapuri, M. Woodacre, and N. Vasseghi. The MIPS R4000 processor. *IEEE Micro*, pages 10–22, April 1992.

[15] Y.-J. Oyang, C.-H. Wen, Y.-F. Chen, and S.-M. Lin. The effects of employing advanced branching mechanisms in superscalar architectures. *ACM Computer Architecture News*, 18(4):35–51, December 1990.

[16] D. A. Patterson and C. H. Séquin. RISC I: A reduced instruction set VLSI computer. In *The Proceedings of the 8th International Symposium on Computer Architecture*, pages 443–457, May 1981.

[17] G. Radin. The 801 minicomputer. In *The Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 39–47, March 1982.

[18] W. F. Richardson and E. L. Brunvand. The NSR processor prototype. Technical Report UUCS-92-029, Department of Computer Science, University of Utah, USA., 1992.

[19] C. L. Seitz. System Timing. In C. Mead and L. Conway, editors, *Introduction to VLSI Systems*, chapter 7, pages 218–262. Addison-Wesley, 1980.

[20] J. E. Smith. A study of branch prediction strategies. In *The Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, May 1981.

[21] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, May 1988.

[22] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[23] N. Ullah and M. Holle. The MC88110 implementation of precise exceptions in a superscalar architecture. *ACM Computer Architecture News*, 21(1):15–25, March 1993.

D.4 Static Scheduling of Instructions on Micronet-based Asynchronous Processors

Title:	Static scheduling of instructions on micronet-based asynchronous processors.
Authors:	D. K. Arvind and **V. E. F. Rebello**.
Presented at:	The 2nd International Symposium on Advanced Research on Asynchronous Circuits and Systems (ASYNC'96).
Place:	Aizu Wakamatsu City, Japan.
Date:	18^{th} – 21^{st} March 1996.
Publisher:	IEEE Computer Society Press.

Static Scheduling of Instructions on Micronet-based Asynchronous Processors

D. K. Arvind and V. E. F. Rebello
Department of Computer Science, The University of Edinburgh
Edinburgh, EH9 3JZ, United Kingdom
E-mail: {dka, vefr}@dcs.ed.ac.uk

Abstract

This paper investigates issues which impinge on the design of static instruction schedulers for micronet-based asynchronous processor (MAP) architectures. The micronet model exposes both temporal and spatial concurrency within a processor. A list scheduling algorithm is described which has been optimised with MAP-specific heuristics. Their performance on some program graphs are presented and conclusions are drawn on the suitability of MAP as targets for ILP compilers.

Keywords: Asynchronous Processor Architecture, Instruction-level Parallelism (ILP), Micronets, Static scheduling.

1 Introduction

A number of novel asynchronous processor architectures have been proposed recently [6, 9, 10, 12, 21, 23, 24, 27], but scant attention has been paid to any understanding of the interactions between the processor and compiler designs. Instead, existing synchronous RISC compiler technology has been reused (largely unmodified), while exploiting any improvements in the performance of the hardware which asynchrony provides.

One of the outcomes of the RISC design approach had been a deeper understanding of the interactions between the processor design and the implementation and compiler technologies, respectively. The processors were streamlined for efficient implementation in the emerging VLSI technology, and the system complexity was migrated upwards to their compilers. For instance, MIPS did away with hardware interlocks and relied instead on the compiler to reorder instructions and introduce null ones where appropriate [15]. The optimisers for synchronous pipelines have assumed a deterministic model of the target, with each stage delay being approximated to being the same, having been fixed *a priori* by the clock. They produce, both, an order of execution for the instructions, and the times - in terms of multiples of the basic RISC instruction cycle, when they are to execute. In contrast, a linear, asynchronous pipeline, e.g. *micropipeline* [28], has stages whose delays can vary, thanks to data dependencies. Now, the compiler has a less accurate timing model of the target, and any optimisations based on a synchronous model, such as scheduling instructions in execution gaps, are less effective.

A *micronet* is a network of pipelines, with (selected) stages of different pipelines being able to communicate with each other. This enables the exploitation of both spatial and temporal concurrency between instructions [2] (in contrast, a micropipeline only exploits temporal parallelism [4]). It is more difficult for a compiler to predict the behaviour of the micronet for the following reasons: firstly, as in a micropipeline the delay of each pipeline stage might vary; secondly and more uniquely, each instruction only visits the relevant stages and the multiple paths enable more than one instruction to operate concurrently within a stage, which enables instructions to race each other, with possible out-of-order completion of instructions. Furthermore, instructions may interfere with each other when competing for the same resource in a particular stage.

The effective performance which a MAP system can deliver depends intimately on the compiler's ability to match the parallelism in programs with the temporal and spatial concurrency exposed by the MAP architecture. This paper is a preliminary attempt to understand the interface between the back-end of a parallelising compiler and MAP architectures. In the rest of this paper, Section 2 briefly describes MAP architectures; Section 3 introduces the MAP schedul-

Published in the Proceedings of the 2nd International Symposium on Advanced Research on Asynchronous Circuits and Systems (ASYNC'96), pp 80-91, Aizu Wakamatsu City, Japan, March 1996.

ing problem, and presents MAP-specific optimisations to a list scheduling algorithm; Section 4 evaluates the quality of the resulting schedules; Section 5 discusses the choice of instruction costs and the effect of instruction interference on the scheduler; and finally, Section 6 provides concluding remarks and scope for future work.

2 A brief introduction to MAP architectures

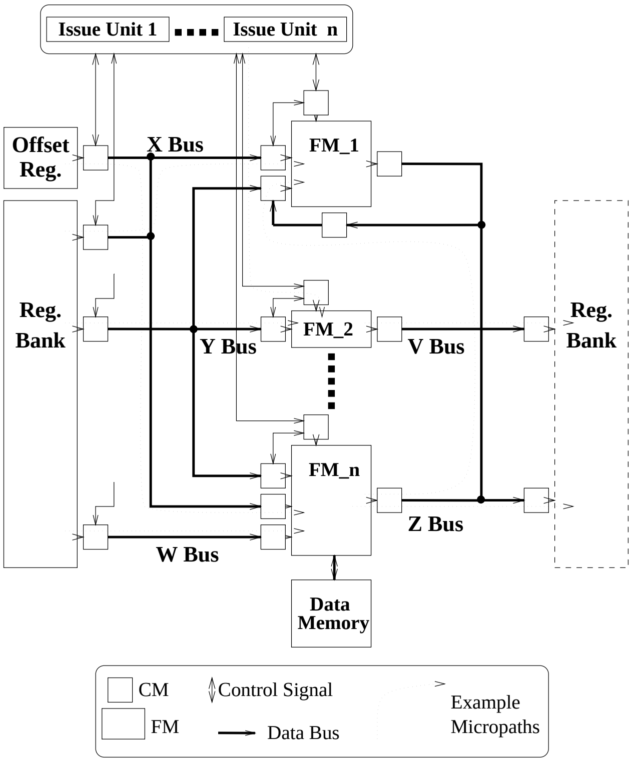


Figure 1: A micronet model of a MAP architecture

A *micronet* is an ensemble of *micropaths*, where a micropath is a pipeline or sequence of *microagents*, and in turn, a microagent performs either a *communicating* or a *functional micro-operation*. A functional microagent (FM) communicates with other FMs through their respective communicating microagents (CM). Even with a single issue unit, where the issue rate is faster than the slowest instruction execution rate, the microagents can all operate concurrently in space, in addition to the temporal concurrency associated with pipelines. Another feature of the micronet is that the micro-operations for an instruction are initiated independently by the issue unit, as soon as their particular microagents become available, and delegates all control to them, thus freeing the issue unit for the next instruction. Therefore, the idle time between instructions is kept to a minimum [2]. The executing instructions also release their microagents individually, as soon as the respective micro-operations have completed, thus freeing the resources immediately for another instruction. Finally, through a novel application of the communication protocol, datapath hazards are resolved efficiently while hiding the overheads of asynchrony [3].

A micronet can be stalled due to contention for resources. In particular, the issue unit will be stalled when the resources required by the current instruction are all busy. The scheduler attempts to minimise this by suitably ordering the instructions at compile-time. If it is impossible to schedule successive unrelated instructions, then the micronet minimises the stall at run-time. In the case of data-dependent instructions: both instructions are issued, with the second instruction awaiting the result to be forwarded. In the case of resource contention: the second instruction performs all the micro-operations up to the microagent which is busy. In effect, only the offending micro-operation is stalled, rather than the entire instruction. A detailed explanation of hazard avoidance is given in [3], with implications for the scheduler being detailed below:

Read-after-Write - Although the dependent instruction will be issued, its execution will be delayed until the completion of its predecessor. In practice, it is preferable not to issue such an instruction, since the resources earmarked for the dependent instruction are unavailable for use by other, now "ready-to-execute", instructions, which might introduce further structural hazards in the bargain.

Write-after-Write - The write-back order has to be maintained and this is achieved in hardware by the micronet. The two instructions are permitted to execute concurrently. Although all of the second instruction's microoperations will have been initiated, the write-back microoperation will stall for as long as the first instruction holds on to the destination register. The current MAP architecture supports only one outstanding register lock request, therefore a subsequent third instruction which requires a locked register cannot be issued, until the first write-back has been completed. The scheduler should avoid arranging instructions which write to the register file immediately after two instructions with write-after-write dependencies.

Write-after-Read - In the case of an architecture with a single set of operand fetch buses, the hardware ensures that a dependent instruction will be unable to lock its destination register before its predecessor has fetched its operand. Should there be a number of operand fetch buses (as in a superscalar MAP), and the possibility of a dependent instruction obtaining its operands before its predecessor, then this instruction may have to be stalled. This would only be necessary when the time to execute the dependent instruction is less than the operand fetch time for the predecessor. This hazard is also known as an anti-dependency, and along with write-after-write hazards can be avoided by register renaming.

Hazard resolution is a good example of the interaction between the compiler and the architecture. Since there is no concept of time in the schedule, it is impossible to avoid all hazards at compile time (c.f. the MIPS organiser). The scheduler can only hope to produce an ordering of instructions which reduces the number of hazards, and relies on the MAP architecture to minimise their effects by efficiently resolving them in hardware.

The computational model for synchronous RISC architectures is simple, in the sense that the execution times of instructions are considered fixed and instructions do not contend for resources. Neither of these hold for MAP architectures. The MAP model describes the architecture as a collection of microagents, where each one has an micro-operation latency which determines when the result of that micro-operation becomes available; and a micro-operation cycle time, which signifies the rate at which the micro-operations can be executed.

3 The MAP Scheduler

The MAP scheduling problem can be stated as follows: *Given a set of heterogeneous resources with variable execution times, devise a minimal-length, non-preemptive schedule which respects dependencies within programs. Each program being described as an arbitrary partial ordering of instructions.*

The precedence- and resource-constrained instruction scheduling problem has been studied well, and it is known that even by imposing restrictions, the problem is still NP-hard [7] [17] [29]. For example, when the execution times of tasks are not uniform and their partial order is arbitrary, then for two or more identical processing units, the problem of determining a minimal-length, non-preemptive schedule is NP-complete [13]. This result is true even if all of the tasks are independent. Therefore, in order to achieve near-optimal execution times for given applications on MAP architectures, an efficient (polynomial-time) scheduling algorithm based on one or a number of heuristics must be devised.

3.1 The MAP Scheduling Problem

List scheduling (LS) is a general method for scheduling tasks in resource- constrained problems [7]. LS builds a *ready set* that contains all of the tasks which are not waiting on the results of other tasks. When a processor becomes available, a task with the highest priority is chosen from the set and assigned to it. The ready set is obtained from a topological sort of the data dependence graph. LS relies on other heuristics to prioritise the ready tasks and guide it towards an optimal solution. This has lead to a profusion of LS-based heuristics [5, 11, 16, 20, 25].

The MAP solution adopted here is based on the optimal, greedy scheduling algorithm for list scheduling which was proposed by Coffman and Graham [8]. This is an $O(n^2)$ algorithm for arbitrary precedence constraints for two processors with unit execution costs. A MAP scheduler has to deal with heterogeneous resources and can no longer just choose the ready instruction with the highest priority, but must also consider whether the correct resources are also available. Once an instruction is issued, its execution cannot be suspended and resumed at the point of suspension at a later time, i.e. schedules must be non-preemptive. The goodness of these schedules are highly dependent on the parameter(s) that are used to prioritise instructions within the ready list [1] [22], and these are next discussed.

3.1.1 Minimising Idle Times

The scheduler's first assumption is that minimising the stall time will lead to an optimal (or at least near-optimal) program execution time (the first priority heuristic). This implies that the MAP compiler should not schedule instructions until the required microagents (resources) are available. Also, the hazards due to data dependencies outlined in the previous section should be avoided. All of this implies that the computational model has to maintain a scoreboard of resource activities.

3.1.2 Primary Instruction Priority

In Coffman and Graham's algorithm, interprocessor communication is assumed to be zero and tasks have unit execution times, which means that time can be conveniently treated as being discrete rather than continuous. This allows priorities to be assigned based on the task's level within the DAG from the sink tasks. Since instructions have different worst-case execution times in MAP, the problem is similar to multiprocessor scheduling with interprocessor communication delays (where communication costs are only incurred if dependent tasks are scheduled on different processors). The solutions adopted in this field have been based on critical path analysis and heuristics [14] [19] [26]. (The critical path cost of a task is the largest sum of costs along a path from itself to a sink task.) In the MAP computational model, although actual instruction execution costs may vary, these critical path costs can be determined *a priori* by basing them on fixed, worst-case instruction costs.

3.1.3 Secondary Instruction Priority

The heuristics applied so far may still not prioritise the executable tasks (i.e. those tasks whose operands and resources are available and are therefore ready for execution) sufficiently. One feature which does seem to significantly influence the best choice of candidate is the dependents of the chosen task. The two heuristics used to "break ties" amongst candidates of the same priority act as follows: the first one gives a higher priority to the task with the larger number of successors which are *solely* dependent on it. If a tie is still unbroken, then a higher priority is given to the task with the most number of successors. A feature of these heuristics is that the priority of a task increases with time. Additionally, these heuristics highlight the need to consider not only which tasks need to execute in the future, but also their resources.

3.1.4 Importance of the Instruction Issue Cycle Time

Unlike synchronous pipelines, micronet resources have two parameters which affect instruction execution costs: the microoperation's latency and its cycle time [4]. Latency determines when data becomes available for subsequent micro-operations and other instructions. The cycle time influences when a resource (microagent) becomes available again for use by a subsequent instruction (or microoperation). Together with program parallelism and the number of resources, a limiting factor on the amount of exploitable ILP is the cycle time of the issue unit in relationship to the execution time of instructions (or more accurately their cycle times).

In order to minimise the issue unit's stall time, the compiler has to devise a schedule that allows instructions to be issued continuously at the highest possible rate, which is equivalent to one every minimum Instruction Issue Cycle Time (IICT) [3]. Traditional synchronous datapaths are pipelined or where necessary super-pipelined (i.e. the functional units are themselves pipelined) sufficiently to achieve this goal. Due to the spatial ILP in MAP, instructions are issued at a rate (determined by the IICT and dependencies) which is faster than their Instruction Cycle Times (ICTs). The ICT is the effective issue time (due to pipelining) for a particular instruction, which is determined by the rate at which that specific instruction type can be processed. As the IICT, which is less than the largest ICT, gets smaller, the MAP architecture behaves more in a superscalar fashion and therefore the value of the IICT itself can have a significant influence on the optimality of a schedule. This is less significant when the IICT is comparable to the largest ICT, in which case the order of the independent instructions is less critical, since the micronet behaves like a linear pipeline without any spatial concurrency.

3.1.5 IICT, ICT and Lookahead

When choosing an instruction to schedule, it may be beneficial to consider not only those instructions which are ready, but also ones which will become ready in the near future, called *instruction lookahead*, e.g. within the next minimum IICT. Note that this may mean deliberately selecting an instruction that causes the processor's issue unit to stall.

Another form of lookahead is to consider the future resource requirements when scheduling instructions, called *resource lookahead*. The two steps in choosing an instruction and checking for availability of resources should take place in conjunction (See Algorithm 1 for more details).

3.2 The MAP scheduling algorithm

The algorithm takes as its input a directed graph of instruction dependencies and a resource graph with architectural parameters, and generates an instruction schedule for the given MAP architecture. Two lists are defined as follows: the WI_list - the list of instructions still awaiting their operands, and the EI_list - an ordered list of instructions which are ready, or will be ready in the near future (for lookahead instructions), but still awaiting issue. The order of the latter list is determined by the critical path costs of instructions, i.e. the primary priority. Next, a prioritised list of executable instructions is derived from the EI_list based on the availability of their resources at the current time. If there are ties, an instruction (or instructions in the case of superscalar MAP) is chosen for issue based on secondary priority values.

The scheduler mimics the behaviour of the architecture's issue unit. The function *generate_schedule()*, as shown in Algorithm 1, schedules instructions based on their readiness, their priority and the availability of resources. Unlike schedulers for synchronous machines, the scheduling of instructions does not proceed in uniform time steps, but rather in an asynchronous event-driven manner until all the instructions are scheduled. Each iteration of the main loop (the while do loop in line 5) corresponds to an instant in time when the issue unit is ready to issue an instruction. However, a situation may arise when at some given time there are no instructions ready for issue (line 8), in which case the clock must be advanced, but only as far as necessary to remedy this. The incrementing of the clock simulates the issue unit being stalled. The routine, *advance_clock()*, finds the earliest occurrence of three types of events: the ready time of an instruction in the WI_list and of a lookahead instruction in the EI_list; the time when the result of an operation becomes available in the register file; and the time a busy resource becomes free. Only the first two events can change the status of the EI_list. There is a choice of heuristics which can be applied, either the instruction lookahead or the traditional priority-based approach. Instruction lookahead (lines 9 - 17) chooses the best instruction to issue from the EI_list based on the lookahead heuristic. The function, *get_ready_instr()*, returns from the given list of instructions the one with the highest estimated-time-to-completion (ETC) priority for which there will be sufficient resources in the datapath if it is issued at its earliest issue time. This time may be the current issue time or some time in the future. In the case of the latter, issuing this instruction will cause the issue unit to stall. In the current implementation of the lookahead heuristic, only one instruction is chosen per issue cycle iteration. The routine, *apply.lookahead()*, implements the instruction lookahead heuristic which uses the ETC priority and the earliest issue time of two instructions to determine which of them should be issued first. By comparing the estimated execution time of the two instruction schedules, the order with the smallest time is chosen. Should the two schedules have the same time, then the order where an instruction completes the earliest is chosen, since this allows dependents to become ready sooner. The alternative heuristic (lines 18 - 29) chooses the instruction with the highest priority which can be issued immediately. This may involve choosing one or more from a number of instructions with the same primary priority value (ETC). Line 19 creates a list of ready instructions with the same, highest ETC values and line 22 removes those instructions with insufficient resources for issue at the current time. Line 23 supports architectures which incorporate lockstep superscalar instruction issue. The routine *issue_all()* issues as many of the instructions as possible from the given list. If there are not enough issue-slots for the complete list (*readyI*), then the routine *choosing_insts()* returns the best instruction for issue based on the secondary priorities. The two loops (lines 26 and 27) repeat until either the issues slots are filled or their respective lists become empty. The clock is advanced appropriately depending on whether or not the scheduler was able to issue one or more instructions at the current time (lines 28 and 29). The routine, *update_writeback*, models the behaviour of the portion of the micronet not directly controlled by the issue unit, e.g. write-back bus. Line 32 updates the instruction lists and the next instruction issue cycle iteration begins at a new time.

Example 1 and Example 2 contrast the influence of IICT and resource lookahead on determining an optimal schedule. A_1 and B are ready candidate instructions, with a third instruction, A_2, which has a structural dependency on A_1.

The lookahead heuristics attempt to match the available program and architectural parallelism over a short window of time. The strategy of repeating the process over the entire program allows the instruction-level parallelism to be exploited more evenly. This has two effects: firstly, a better program makespan is usually achieved; secondly, a schedule is generated which is more robust to deviations from the predicted instruction costs because only the appropriate amount of program parallelism is exposed which can be exploited by the target at any one time. Since costs are based on worse-case values rather than typical ones, the traditional list scheduling heuristics tend to overly migrate independent instructions to the top of the schedule, leaving insufficient parallelism for later. Kerns and Eggers [18] proposed a code scheduling algorithm called *balanced scheduling* for synchronous architectures which is similar in concept. Their algorithm is specifically designed to tolerate a wide range of variance in load latency, e.g. cache misses/hits, global and

Algorithm 1 : ***The MAP scheduler (generate_schedule())***

```
curr_time := 0;
calc_completion_times(); \* Critical path analysis for each instruction *\
update_WI(WI_list); \* Determine instruction start times *\
update_EI(WI_list); \* Move ready instructions to EI_list *\
while (WI_list ≠ {}) or (EI_list ≠ {}) do
    no_issued := 0; \* Number of inst issued simultaneously at this time *\
    candidates := EI_list;
    if (EI_list = {}) curr_time := advance_clock(YES, YES, NO, curr_time);
    else if (lookahead = YES) \* Use Instruction Lookahead Heuristics *\
        BestChoice := get_ready_instr(candidates); \* The inst with the highest *\
        \* priority in the candidates list for which there are sufficient resources *\
        if (BestChoice ≠ NULL)
            while candidates ≠ {} do
                NextInst := get_ready_instr(candidates);
                if (NextInst ≠ NULL) apply.lookahead(BestChoice, NextInst);
            end while
            if (BestChoice.rdy_time ≤ curr_time + issue_cost)
                issue_instruction(BestChoice); no_issued++;
                EI_list := EI_list - BestChoice;

    else
        do \* Alternative strategy without Instruction Lookahead *\
            \* Let same_ETC_list be the list of the highest ETC cost ready insts *\
            ∃ same_ETC_list ⊆ candidates, s.t. ∀ i ∈ candidates,
                ∃ v ∈ same_ETC_list, s.t. (v.ETC ≥ i.ETC);
            candidates := candidates - same_ETC_list;
            do \* Remove instructions without sufficient resources *\
                ∃ readyI ⊆ same_ETC_list, s.t. ∀i ∈ readyI,
                    find_avail_FU_resources(i, datapath, curr_time);
                    if (|readyI| ≤ spsclr_deg - no_issued) issue_all(readyI, no_issued);
                    else \* choose between insts in readyI list *\
                        inst_chosen := choosing_insts(readyI, no_issued);
                        EI_list := EI_list - {inst_chosen};

            while ((no_issued < spsclr_deg) and (same_ETC_list ≠ {}));
        while ((no_issued < spsclr_deg) and (candidates ≠ {}));
        if (no_issued > 0) curr_time += inst_issue_cycle;
        else curr_time := advance_clock(YES, YES, YES, curr_time);
    end if
    update_writeback(datapath);
    if (WI_list ≠ {})
        update_WI(WI_list); update_EI(WI_list);
end while
update_writeback(datapath);
```

Example 1 : ***Resource Lookahead***

switch IICT
  case 0: *Choose schedule* $\{A_1,B,A_2\}$ *or* $\{B,A_1,A_2\}$;
    \∗ *Either schedule is optimal* ∗\
  case $(0 \leq \mathrm{IICT} < \frac{1}{2}\cdot\mathrm{ICT}_A)$:
    if $(\mathrm{ICT}_B > 2\cdot\mathrm{ICT}_A)$ *Choose schedule* $\{B,A_1,A_2\}$;
      \∗ *Instruction B takes longer than the both* A_1 *and* A_2 ∗\
    else *Choose schedule* $\{A_1,B,A_2\}$;
    \∗ *In other words, combine the resource requirements of* ∗\
    \∗ *dependent instructions and schedule the instruction* ∗\
    \∗ *according to the resource with the most work.* ∗\
  case $(\frac{1}{2}\cdot\mathrm{ICT}_A \leq \mathrm{IICT} < \mathrm{ICT}_A)$:
    if $(\mathrm{ICT}_B > 2\cdot\mathrm{ICT}_A)$ \∗ *then schedule* B *first (as before)* ∗\
      Choose schedule $\{B,A_1,A_2\}$;
    else \∗ *schedule* A_1 *first*∗\
      if $(\mathrm{ICT}_B < \mathrm{ICT}_A)$ *Choose schedule* $\{A_1,A_2,B\}$;
    else *Choose schedule* $\{A_1,B,A_2\}$;
  case $(\mathrm{ICT}_A \leq \mathrm{IICT})$:
    \∗ *Schedule the instruction with the largest* ICT *first* ∗\
    if $\mathrm{ICT}_A < \mathrm{ICT}_B$ *Choose schedule* $\{B,A_1,A_2\}$;
    else *Choose schedule* $\{A_1,A_2,B\}$;
end switch;

Example 2 : ***Without Resource Lookahead***

if $(\mathrm{IICT} = 0)$ *Choose schedule* $\{A_1,B,A_2\}$ *or* $\{B,A_1,A_2\}$;
  \∗ *Again, either schedule is optimal* ∗\
else \∗ *Simply schedule the instruction with the largest* ICT *first.* ∗\
  if $(\mathrm{ICT}_A < \mathrm{ICT}_B)$
    Choose schedule $\{B,A_1,A_2\}$;
  else if $(\mathrm{IICT} < \mathrm{ICT}_A)$
    Choose Schedule$\{A_1,B,A_2\}$;
  else *Choose schedule* $\{A_1,A_2,B\}$;

local memory. In these architectures, instruction costs are well defined and considered fixed. Usually the latencies reflect the most optimistic execution, e.g., the time of a cache hit rather than a cache miss. Traditional schedulers improve performance through reordering instructions to avoid pipeline stalls, e.g., by inserting independent instructions after loads to keep the CPU busy. The number of instructions inserted (in the best case) depends on this latency value. If the load instruction is delayed beyond the scheduler's estimate, then the processor will stall. However, if the latency is shorter, then the destination register of the load instruction will be tied up for longer and this may increase register pressure enough to cause unnecessary code spills. Both balanced scheduling and resource lookahead are computationally more expensive than the traditional list scheduling approach, and will not be considered further in this paper.

4 Results

In this section, the makespans of MAP schedules for a number of typical instruction DAGs (briefly described below) are compared with their optimal. (The optimal makespan is derived from an exhaustive search.)

BT3 - A Binary Tree with three levels.

BT3.5 - A Binary Tree with three and half levels.

BT4 - A Binary Tree with four levels.

DD - Diamond DAGs which are commonly found in the evaluation of partial differential equations.

DM - Dense matrix multiplication.

SM - Sparse matrix multiplication.

CC - Mix of Load, Store and ALU instructions with data dependencies.

CCL - A loop unrolled version of CC.

Min1 - This architecture contains the minimum resources - one ALU and one Memory Unit (MU) which both share a single write-back bus. The cycle times and latencies of the ALU and MU micro-operations are assumed to the same.

3bus - This architecture has an additional ALU and each of the three functional units has a dedicated write-back bus. (The microoperation cycle times and latencies are the same as Min1).

Min2 - Same as Min1, except that the microoperation costs of the ALU and MU are different and reflect realistic costs obtained from SPICE-level simulations.

The results for the MAP scheduling heuristic, both without and with instruction lookahead, are shown in Table 1. For each DAG, the number of valid schedules is recorded together with the optimal makespan for the given target architecture. The makespan generated by the heuristics together with its closeness to the optimal (recorded both as a percentage of the optimal (*% Diff*) and as a percentage of the difference between the best and worst makespans (*% of the Range*) are also included. It is assumed that sufficient registers are available and so code spilling could be avoided. This would normally be determined at the register allocation phase of the compilation and is not considered here.

The results look quite promising. In a majority of the cases for the **3bus** architecture, the heuristic can find an optimal solution (only in the case of SM is instruction lookahead required to reduce the makespan to optimal). However, the MAP scheduler does not seem to do as well on the **Min1** architecture (for BT3, BT3.5, BT4, CCL, DM and SM). The reason for the poorer makespans is due to a bottleneck on the write-back bus. It turns out to be better in some cases to stall the issue unit for a longer period of time than that assumed by instruction lookahead (the IICT), i.e. wait until a higher priority instruction becomes ready, because this stall time is hidden by the write-back bottleneck. Where the makespan is only slightly worse than the optimal, i.e. DM, the heuristic together with instruction lookahead is sufficient to find an optimal solution. In the case of the **Min2** architecture, BT3, BT3.5, BT4, and CCL are now optimal. This is because the relative delays of the microagents have reduced the bottleneck for the write-back bus. In the case of DM and SM, there is still interference between the instructions which result in sub-optimal executions. Instruction interference can be reduced by applying a post-pass re-ordering of the generated schedules, and this is the subject of a future paper.

Remember that these schedules are only optimal with respect to the instructions costs which have been assumed. In practice, these schedules may not be optimal for a particular execution of the program for the reasons discussed earlier. One could even expect that each run of the program would have a different optimal schedule. The stability of the schedules in light of variance in the resource delays needs further study.

Prgm DAG	MAP Arch	No. of Valid Schds	No. of Optimal Schds	The Optimal Mkspn	*The MAP Heuristic*			*MAP with Lookahead*		
					Make-span	% Diff	% of the Range	Make-span	% Diff	% of the Range
BT3	Min1	640	24	1105nS	1185nS	92.76%	75%	1185nS	92.76%	75%
BT3.5	Min1	230400	512	1505nS	1585nS	94.68%	85.71%	1585nS	94.68%	85.71%
BT4	Min1	21964800	529920	1785nS	1885nS	94.4%	85.71%	1885nS	94.4%	85.71%
DD	Min1	42	2	1325nS	1325nS	100%	100%	1325nS	100%	100%
DM	Min1	310160	200	1905nS	1925nS	98.95%	98.11%	1905nS	100%	100%
SM	Min1	46574	24	2085nS	2245nS	92.33%	81.81%	2265nS	91.37%	79.55%
CC	Min1	4	2	735nS	735nS	100%	100%	735nS	100%	100%
CCL	Min1	4032	4	945nS	1015nS	92.59%	88.89%	1015nS	92.59%	88.89%
BT3	3bus	640	72	1105nS	1105nS	100%	100%	1105nS	100%	100%
BT3.5	3bus	230400	128	1355nS	1355nS	100%	100%	1355nS	100%	100%
BT4	3bus	21964800	456960	1605nS	1605nS	100%	100%	1605nS	100%	100%
DD	3bus	42	2	1225nS	1225nS	100%	100%	1225nS	100%	100%
DM	3bus	310160	156	1645nS	1645nS	100%	100%	1645nS	100%	100%
SM	3bus	46574	46	2005nS	2035nS	99%	97.67%	2005nS	100%	100%
CC	3bus	4	2	735nS	735nS	100%	100%	735nS	100%	100%
CCL	3bus	4032	18	835nS	835nS	100%	100%	835nS	100%	100%
BT3	Min2	640	32	930nS	930nS	100%	100%	930nS	100%	100%
BT3.5	Min2	230400	704	1230nS	1230nS	100%	100%	1230nS	100%	100%
BT4	Min2	21964800	768768	1500nS	1500nS	100%	100%	1500nS	100%	100%
DD	Min2	42	2	570nS	570nS	100%	100%	570nS	100%	100%
DM	Min2	310160	120	1250nS	1280nS	97.6%	92.5%	1250nS	100%	100%
SM	Min2	46574	2	1180nS	1200nS	98.3%	95.9%	1190nS	99.15%	97.96%
CC	Min2	4	2	400nS	400nS	100%	100%	400nS	100%	100%
CCL	Min2	4032	2	550nS	550nS	100%	100%	550nS	100%	100%

Table 1: Measuring the optimality of the scheduling heuristics

The Scheduling Costs

	Worst-case Costs	Average-case Costs
Issue Cycle Time	3	2
ALU Instruction	4	3
Load Instruction	10	4
Store Instruction	6	3

The Schedule Based on Worst-Case Costs			**The Schedule Based on Average-case Costs**		
Instruction Schedule	Execution Time using worst-case run-time costs	Execution Time using average-case run-time costs	Instruction Schedule	Execution Time using worst-case run-time costs	Execution Time using average-case run-time costs
LD	0 - 10	0 - 4	ALU	0 - 4	0 - 3
ALU	3 - 7	2 - 5	LD	3 - 13	2 - 6
ALU	6 - 11	4 - 7	ALU	6 - 10	4 - 7
ST	10 - 16	7 - 10	ST	9 - 19	6 - 9
ALU	13 - 17	9 - 12	ALU	16 - 20	8 - 11
Execution Times : 17		12		20	11

Figure 2: The makespans of schedules based on worst- and average-case run-time costs

5 Discussion

5.1 Choice of instruction execution costs

Although the execution times of the same instruction might vary due to data-dependent delays, worst-, average- or even best-case figures for the execution times can be found on which the schedules could be based. When producing static schedules, the compiler has to use the delays of the FMs and the question arises as to which of the sets of figures to use. Figure 2 illustrates the simplified schedules for the CC test (obtained from [15]) based on worst-case and average-case costs and figures for the execution times of the instructions based on actual worst-case and average-case delays at run-time for these schedules. (The ratios of the delays for the two cases for the instructions realistically reflect actual behaviour on the asynchronous processor under study.) The figures reveal that given these ratios, using a schedule based on worst-case costs is better in practice. Using this approach a heuristic will always try to schedule an instruction, if possible, only when its operands are guaranteed to be available, thereby minimising any stalls. Note also that the schedule's correctness is not affected by the changes in instruction costs. Furthermore, given that a program's critical path may change with different executions (due to different data sets) and that the schedule is generated once, the compiler's choice of which costs to use is important. By basing the schedule on worst-case delays a lower bound on performance can be achieved.

5.2 Interaction between executing instructions

Reasons other than the ones just stated also contribute to the difficulty in predicting the global state of the micronet. In synchronous processors, the compiler can assume when scheduling a basic block that the datapath is idle and that all of the resources are available. This is a consequence of the fact that in synchronous pipelines, an instruction never affects the execution of other instructions. This is not necessarily the case in a micronet, since the execution times of instructions might vary for the following reasons: only a partial ordering is employed between instructions (i.e. it is not necessary for the previous instructions to have completed their execution before successive ones); instructions compete for shared resources, e.g. the write-back bus; during execution instructions might interfere with each other. Therefore, the state of the resources at any particular time cannot be predicted accurately at compile-time. But this information is indeed available at run-time in the issue unit of the micronet. This could be used to dynamically tune (i.e. allow out-of-order instruction issues) the static schedule by the control unit. This requires identifying an instruction which can be executed immediately (easily achieved using the control acknowledgement signal scoreboarding mechanism), and checking that the instruction is independent of earlier ones in the instruction buffer. Although the latter may be expensive to perform, the task can be made simpler with assistance from the compiler by using a concurrency bit.

6 Conclusions

The MAP approach efficiently combines aspects of well-known architectural styles. In dataflow architectures, the instructions are issued as soon as their operands are available. This is achieved dynamically in hardware which incurs not insignificant run-time (book-keeping) costs. As in RISC architectures, code scheduling is done statically, but additionally instruction issue (and even possibly the instruction schedule) is fine-tuned dynamically to take advantage of run-time characteristics as in the data-flow model. In a sense, at the instruction-level MAP follows the classical von Neumann style, whereas at the level of microagents it is more in the character of dataflow architectures.

The micronet model exposes temporal and spatial concurrency in the datapath, with fine-grained resources now being visible to the compiler. This model subsumes the micropipeline model which only exploits temporal concurrency in the datapath and the scheduling methods described here can be equally applied to micropipeline-based processors.

Code scheduling (on ILP architectures) and machine-dependent optimisations have a significant impact on program performance. It is the task of the compiler to schedule instructions such that these resources are efficiently utilised. The instruction schedule is devised based on a computational model of the target architecture. For synchronous architectures the model is simple; in contrast, an asynchronous model is necessarily less accurate for the reasons discussed earlier. However, initial studies have shown that these factors do not significantly hinder a compiler's ability to schedule code efficiently. Worst-case instruction execution times have been considered and where the resulting schedule is treated as a first pass one. The interference between the instructions can be reduced by applying post-pass optimisations, and this is being currently investigated. The instructions could then be dynamically reordered at run-time to fine-tune this schedule by taking advantage of actual run-time costs. In conclusion, preliminary studies have shown that a micronet-based asynchronous processor architecture does present a suitable target for an ILP compiler.

Acknowledgements

V. Rebello was supported by a postgraduate studentship from the U. K. Engineering and Physical Sciences Research Council (EPSRC). This work was partially supported by a grant from EPSRC entitled *Formal Infusion of Communication and Concurrency into Programs and Systems* (Grant Number GR/G55457).

References

[1] T. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, December 1978.

[2] D. K. Arvind, R. D. Mullins, and V. E. F. Rebello. Micronets: A model for decentralising control in asynchronous processor architectures. In M. B. Josephs, editor, *The Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, pages 190–199, London, UK, May 1995. IEEE Computer Society Press.

[3] D. K. Arvind and V. E. F. Rebello. Instruction-level parallelism in asynchronous processor architectures. In M. Moonen and F. Catthoor, editors, *Proceedings of the 3rd International Workshop on Algorithms and Parallel VLSI Architectures*, pages 203–215, Leuven, Belgium, August 1994. Elsevier Science Publishers.

[4] D. K. Arvind and V. E. F. Rebello. On the performance evaluation of asynchronous processor architectures. In P. Dowd and E. Gelenbe, editors, *Proceedings of the 3rd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'95)*, pages 100–105, Durham, NC, USA, January 1995. IEEE Computer Society Press.

[5] J. Baxter and J. H. Patel. The LAST Algorithm: A heuristic-based static task allocation algorithm. In *The Proceedings of the 1989 International Conference on Parallel Processing*, pages 217–222, 1989.

[6] E. Brunvand. The NSR processor. In *The Proceedings of the Hawaii International Conference on System Sciences*. IEEE Computer Society Press, January 1993.

[7] E. G. Coffman. *Computer and Job-Shop Scheduling Theory*. John Wiley and Sons, New York, 1976.

[8] E. G. Coffman and R. L. Graham. Optimal scheduling for two-processor systems. *Acta. Informatica*, 1:200–213, 1972.

[9] I. David, R. Ginosar, and M. Yoeli. Self-timed architecture of a reduced instruction set computer. In S. Furber and M. Edwards, editors, *The Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies*, Manchester, UK, March 1993. Elsevier Science Publishers.

[10] Mark E. Dean. *STRiP: A Self-timed RISC Processor*. PhD thesis, Stanford University, July 1992.

[11] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9:138–153, 1990.

[12] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In T. Yanagawa and P. A. Ivey, editors, *The Proceedings of the IFIP International Conference on Very Large Scale Integration (VLSI'93)*, pages 5.4.1–5.4.10, Grenoble, France, September 1993.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[14] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16:276–291, December 1992.

[15] J. Hennessy and T. Gross. Postpass code optimisation of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[16] J-J. Hwang, Y-C. Chow, F. D. Anger, and C-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal of Computing*, 18(2):244–257, April 1989.

[17] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, C-33(11):1023–1029, November 1984.

[18] D. R. Kerns and S. J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. *SIGPLAN Notices*, 28(6):278–289, June 1993. *Proceedings of the ACM Conference on Programming Language Design and Implementation.*

[19] S. J. Kim and J. C. Brown. A general approach to mapping of parallel computation upon multiprocessor architecture. In *The Proceedings of the International Conference on Parallel Processing, Vol. III*, pages 1–8, 1988.

[20] S. Manoharan and P. Thanisch. Assigning dependency graphs onto processor networks. *Parallel Computing*, 17(1):63–73, April 1991.

[21] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus. The design of an asynchronous microprocessor. In C. L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 351–373, Cambridge, Mass., 1989. MIT Press.

[22] C. McCreary, A. A. Khan, J. Thompson, and M. E. McArdle. A comparison of heuristics for scheduling DAGs on multiprocessors. Technical Report CSE-93-07, Auburn University, Auburn, AL, 36849. USA., 1994.

[23] S. V. Morton, S. S. Appleton, and M. J. Liebelt. ECSTAC: A fast asynchronous microprocessor. In M. B. Josephs, editor, *The Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, pages 180–189, London, UK, May 1995. IEEE Computer Society Press.

[24] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura. TITAC: Design of a quasi-delay-insensitive microprocessor. *IEEE Design and Test of Computers*, pages 50–63, Summer 1994.

[25] C. H. Papadimitrou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal of Computing*, 19(2):322–328, April 1990.

[26] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* The MIT Press, 1989.

[27] R. F. Sproull, I. E. Sutherland, and C. E. Molnar. Counterflow pipeline processor architecture. Technical Report SMLI TR-94-25, Sun Microsystems Laboratories Inc., April 1994.

[28] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[29] J. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10:384–393, 1975.

Bibliography

[1] T. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, December 1978.

[2] M. Afghahi and C. Svennson. Performance of synchronous and asynchronous schemes for VLSI systems. *IEEE Transactions on Computers*, 41(7):858–872, July 1992.

[3] A. Aiken and A. Nicolau. A development environment for horizontal microcode. *IEEE Transactions on Software Engineering*, 14(5):584–594, May 1988.

[4] D. K. Arvind, R. D. Mullins, and V. E. F. Rebello. Micronets: A model for decentralising control in asynchronous processor architectures. In M. B. Josephs, editor, *The Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, pages 190–199, London, UK, May 1995. IEEE Computer Society Press.

[5] D. K. Arvind and V. E. F. Rebello. Instruction-level parallelism in asynchronous processor architectures. In M. Moonen and F. Catthoor, editors, *Proceedings of the 3rd International Workshop on Algorithms and Parallel VLSI Architectures*, pages 203–215, Leuven, Belgium, August 1994. Elsevier Science Publishers.

[6] D. K. Arvind and V. E. F. Rebello. On the performance evaluation of asynchronous processor architectures. In P. Dowd and E. Gelenbe, editors, *Proceedings of the 3rd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'95)*, pages 100–105, Durham, NC, USA, January 1995. IEEE Computer Society Press.

[7] D. K. Arvind and V. E. F. Rebello. Static scheduling of instruction on micronet-based asynchronous processors. In *The Proceedings of the 2nd International Symposium on Advanced Research on Asynchronous Circuits and Systems (ASYNC'96)*, pages 80–91, Aizu Wakamatsu City, Japan, March 1996. IEEE Computer Society Press.

[8] D. K. Arvind and C. R. Smart. A unified framework for parallel event-driven logic simulation. In *Proceedings of the 1991 Computer Simulation Conference*, Baltimore, Maryland, USA, July 1991.

[9] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[10] H. G. Baker. Precise instruction scheduling without a precise machine model. *ACM Computer Architecture News*, 19(6):4–8, December 1991.

[11] M. R. Barbacci and D. P. Siewiorek. *The Design and Analysis of Instruction Set Processors*. McGraw Hill, 1982.

[12] J. Baxter and J. H. Patel. The LAST Algorithm: A heuristic-based static task allocation algorithm. In *The Proceedings of the 1989 International Conference on Parallel Processing*, pages 217–222, 1989.

[13] D. Bernstein, D. Cohen, Y. Lavon, and V. Rainish. Performance evaluation of instruction scheduling on the IBM RISC System/6000. In *The Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO'25)*, pages 226–235, 1992.

[14] D. Bernstein and I. Gertner. Scheduling expressions on a pipelined processor with a maximal delay of one cycle. *ACM Transactions on Programming Languages and Systems*, 11(1):57–66, 1989.

[15] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *The Proceedings of the Conference on Programming Language Design and Implementation*, pages 241–255, June 1991.

[16] B. Bose and T. R. N. Rao. Theory of unidirectional error correcting/detecting codes. *IEEE Transactions on Computers*, C-31(6):521–530, June 1982.

[17] D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating register allocation and instruction scheduling for RISCs. *ACM Computer Architecture News*, 19(2):122–131, April 1991.

[18] E. Brunvand. The NSR processor. In *The Proceedings of the Hawaii International Conference on System Sciences*. IEEE Computer Society Press, January 1993.

[19] E. Brunvand and R. F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *The Proceedings of the International Conference on Computer Aided Design (ICCAD-89)*, pages 262–265, November 1989.

[20] J. A. Brzozowski and J. C. Ebergen. Recent developments in the design of asynchronous circuits. In *Fundamentals of Computation Theory*, pages 78–94. Lecture Notes in Computer Science, Vol. 380, Springer-Verlag, 1989.

[21] J. A. Brzozowski and J. C. Ebergen. On the delay-sensitivity of gate networks. *IEEE Transactions on Computers*, 41(11):1349–1359, November 1992.

[22] J. A. Brzozowski and K. Raahemifar. Testing C-elements is not elementary. In M. B. Josephs, editor, *The Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, pages 150–159, London, UK, May 1995. IEEE Computer Society Press.

[23] W. Buchholz. *Planning a Computer System: Project Stretch*. McGraw-Hill, 1962.

[24] J. Bunda, W. C. Athas, and D. Fussel. Evaluating power implications of cmos microprocessor design decisions. In *The Proceedings of the 1994 International Workshop on Low Power Design*, pages 147–152, Napa, CA, USA., 1994.

[25] S. M. Burns. Automated compilation of concurrent programs into self-timed circuits. Technical Report Caltech-CS-TR-88-2, Computer Science Department, California Institute of Technology, 1988.

[26] S. M. Burns. *Performance Analysis and Optimisation of Asynchronous Circuits*. PhD thesis, Computer Science Department, California Institute of Technology, Pasadena, California, USA, 1991.

[27] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation and spilling via graph coloring. *Computer Languages*, 6:47–57, 1981.

[28] T. J. Chaney and C. E. Molnar. Anomalous behaviour of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, 22(4):421–422, April 1973.

[29] C.-H. Chien, M. A. Franklin, T. Pan, and P. Prabhu. ARAS: Asynchronous RISC architecture simulator. In M. B. Josephs, editor, *The Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, pages 210–219, London, UK, May 1995. IEEE Computer Society Press.

[30] K. M. Chu and D. I. Pulfrey. Design procedures for differential cascade voltage switch circuits. *IEEE Journal of Solid-State Circuits*, 21(6):1082–1087, 1986.

[31] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, June 1987.

[32] E. G. Coffman. *Computer and Job-Shop Scheduling Theory*. John Wiley and Sons, New York, 1976.

[33] E. G. Coffman and R. L. Graham. Optimal scheduling for two-processor systems. *Acta. Informatica*, 1:200–213, 1972.

[34] R. P. Colwell, C. Y. Hitchcock, E. D. Jensen, H. M. B. Sprunt, and C. P. Kollar. Computers, complexity and controversy. *IEEE Computer*, 18:8–19, September 1985.

[35] I. David, R. Ginosar, and M. Yoeli. An efficient implementation of boolean functions as self-timed circuits. *IEEE Transactions on Computers*, 41(1):2–11, January 1992.

[36] I. David, R. Ginosar, and M. Yoeli. Implementing sequential-machines as self-timed circuits. *IEEE Transactions on Computers*, 41(1):12–17, January 1992.

[37] I. David, R. Ginosar, and M. Yoeli. Self-timed architecture of a reduced instruction set computer. In S. Furber and M. Edwards, editors, *The Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies*, Manchester, UK, March 1993. Elsevier Science Publishers.

[38] M. E. Dean, D. L. Dill, and M. Horowitz. Self-timed logic using current-sensing completion detection (CSCD). In *The Proceedings of the International Conference on Computer Design (ICCD'91)*, pages 187–191. IEEE Computer Society Press, October 1991.

[39] Mark E. Dean. *STRiP: A Self-timed RISC Processor*. PhD thesis, Stanford University, July 1992.

[40] K. Diefendorff and M. Allen. Organisation of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, 12(2):40–63, April 1992.

[41] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[42] D. W. Dobberpuhl *et al.* A 200-MHz 64-bit dual issue CMOS processor. *IEEE Journal of Solid-State Circuits*, 27(11):1555–1567, November 1992.

[43] J. C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, 1991.

[44] J. H. Edmondson, P. Rubinfeld, R. Preston, and V. Rajagopalan. Superscalar instruction execution in the 21164 Alpha microprocessor. *IEEE Micro*, 15(2):33–43, April 1995.

[45] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9:138–153, 1990.

[46] J. R. Ellis. *Bulldog: A Compiler for VLSI Architectures*. MIT Press, 1986. PhD Thesis, Yale, 1985.

[47] C. J. Elston, D. B. Christianson, P. A. Findlay, and G. B. Steven. Hades - Towards the design of an asynchronous superscalar processor. In M. B. Josephs, editor, *The Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, pages 200–209, London, UK, May 1995. IEEE Computer Society Press.

[48] P. Endecott. Processor architectures for power efficiency and asynchronous implementation. Master's thesis, Department of Computer Science, University of Manchester, UK., 1993.

[49] P. Endecott. *SCALP: A Superscalar Asynchronous Low-Power Processor*. PhD thesis, Department of Computer Science, University of Manchester, UK., December 1995. CST-41-86.

[50] European Silicon Structures Limited. *Solo 1400 Reference Manual*. ES2 Publications Unit, Bracknell, U.K., 1990.

[51] C. Farnsworth, D. A. Edwards, J. Lie, and S. S. Sikand. A hybrid asynchronous system design environment. In M. B. Josephs, editor, *The Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, pages 91–98, London, UK, May 1995. IEEE Computer Society Press.

[52] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[53] M. J. Flynn, C. L. Mitchell, and J. M. Mulder. And now a case for more complex instruction sets. *IEEE Computer*, 20(9):71–83, September 1987.

[54] M. A. Franklin and T. Pan. Clocked and asynchronous instruction pipelines. In *The Proceedings of the 26th Annual International Symposium on Microarchitecture (MICRO'26)*, pages 177–184, Austin, Texas, USA, December 1993. IEEE Computer Society Press.

[55] S. B. Furber. Lessons from AMULET1: Towards AMULET2. In *Computing Without Clocks: Asynchronous Microprocessor Design*. The Sun Annual

Lecture in Computer Science at the University of Manchester, September 1994.

[56] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In T. Yanagawa and P. A. Ivey, editors, *The Proceedings of the IFIP International Conference on Very Large Scale Integration (VLSI'93)*, pages 5.4.1–5.4.10, Grenoble, France, September 1993.

[57] H. Gabow. An almost linear algorithm for two-processor scheduling. *Journal of the ACM*, 29(3):766–780, 1982.

[58] G. R. Gao. An efficient hybrid dataflow architecture. *Journal of Parallel and Distributed Computing*, 19:293–307, 1993.

[59] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[60] J. D. Garside. A CMOS VLSI implementation of an asynchronous ALU. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 181–207. Elsevier Science Publishers, 1993.

[61] J.-L. Gaudiot and L. Bic. *Advanced Topics in Dataflow Computing*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1991.

[62] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16:276–291, December 1992.

[63] R. Ginosar and N. Michell. On the potential of asynchronous pipelined processors. *ACM Computer Architecture News*, 18(4):27–34, December 1990.

[64] G. Gopalakrishnan. Some unusual micropipeline circuits. Technical Report UUCS-93-015, Department of Computer Science, University of Utah, Salt Lake City, UT, USA, December 1993.

[65] W. R. Hamburgen and J. S. Fitch. Packaging a 150W bipolar ECL microprocessor. Research report 92/1, DEC Western Research Laboratory, March 1992.

[66] S. Hauck. Asynchronous design methodologies: An overview. Technical Report TR 93-05-07, Department of Computer Science and Engineering, University of Washington, Seattle, USA, 1993.

[67] P. Hazewindus. *Testing Delay-Insensitive Ciruits*. PhD thesis, California Institute of Technology, Pasadena, CA, USA., 1992. CS-TR-92-14.

[68] S. Heath. *VMEbus User's Handbook*. CRC Press, 1988.

[69] L. G. Heller and W. R. Griffin. Cascade Voltage Switch Logic: A differential CMOS logic family. In *The Proceedings of the IEEE International Conference on Solid-state Circuits*, pages 16–17, 1984.

[70] J. Hennessy and T. Gross. Postpass code optimisation of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[71] J. Hennessy, N. Jouppi, F. Baskett, and J. Gill. MIPS: A VLSI processor architecture. In *The Proceedings of the CMU Conference on VLSI Systems and Computations*, Rockville, Md. USA., October 1981. Computer Science Press.

[72] J. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[73] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[74] L. A. Hollaar. Direct implementation of asynchronous control units. *IEEE Transactions on Computers*, C-31(12):1133–1141, December 1982.

[75] T. C. Hu. Parallel sequencing and assembly line problems. *Operational Research*, 9(6):841–848, 1961.

[76] H. Hulgaard, S. M. Burns, and G. Borriello. Testing asynchronous circuits: A survey. Technical Report Technical Report UW-CSE-94-03-06, Department of Computer Science and Engineering, University of Washington, 1994.

[77] J-J. Hwang, Y-C. Chow, F. D. Anger, and C-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal of Computing*, 18(2):244–257, April 1989.

[78] INMOS Limited. *Occam2 Reference Manual*. Prentice Hall International, 1988.

[79] INMOS Limited. *Transputer Reference Manual*. Prentice Hall International, 1988.

[80] R. Jain. *The Art of Computer System Performance Analysis*. John Wiley & Sons, 1991.

[81] M. Johnson. *Superscalar Processor Design*. Prentice-Hall, Englewood Cliffs, NJ, USA., 1991.

[82] M. B. Josephs and J. T. Udding. Delay-insensitive circuits: An algebraic approach to their design. In J. C. M. Baeten and J. W. Klop, editors, *Theories of Concurrency: Unification and Extension (CONCUR'90)*, pages 342–366. Springer-Verlag, August 1990.

[83] N. P. Jouppi, P. Boyle, and J. S. Fitch. Designing, packaging and testing a 300-Mhz, 115W ECL microprocessor. *IEEE Micro*, 14(2):50–58, April 1994.

[84] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *The Proceedings of ASPLOS III*, pages 272–282. ACM Press, April 1989.

[85] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, C-33(11):1023–1029, November 1984.

[86] M. G. Katevenis, R. W. Sherbourne, D. A. Patterson, and C. H. Séquin. The RISC II micro-architecture. In F. Anceau and E. J. Aas, editors, *The Proceedings of VLSI'83: VLSI Design of Digital Systems*, pages 349–359. North-Holland, 1983.

[87] D. Kearney and N. W. Bergmann. Performance evaluation of asynchronous logic pipelines with data dependent processing delays. In M. B. Josephs, editor, *The Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, pages 4–13, London, UK, May 1995. IEEE Computer Society Press.

[88] D. R. Kerns and S. J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. *SIGPLAN Notices*, 28(6):278–289, June 1993. *Proceedings of the ACM Conference on Programming Language Design and Implementation*.

[89] R. W. Keyes. The evolution of digital electronics towards VLSI. *IEEE Transactions on Electronic Devices*, ED-26(4):271–278, 1979.

[90] A. Khoche and E. Brunvand. Testing self-timed circuits using partial scan. In M. B. Josephs, editor, *The Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, pages 160–169, London, UK, May 1995. IEEE Computer Society Press.

[91] S. J. Kim and J. C. Brown. A general approach to mapping of parallel computation upon multiprocessor architecture. In *The Proceedings of the International Conference on Parallel Processing, Vol. III*, pages 1–8, 1988.

[92] M. Ko. Instruction scheduling for micronet-based asynchronous processors. Master's thesis, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland, UK., September 1995.

[93] S. Komori, H. Takata, T. Tamura, F. Asai, T. Ohno, O. Tomisawa, T. Yamasaki, K. Shima, K. Asada, and H. Terada. An elastic pipeline mechanism by self-timed circuits. *IEEE Journal of Solid-State Circuits*, 23(1):111–117, February 1988.

[94] R. F. Krick and A. Dollas. The evolution of instruction sequencing. *IEEE Computer*, 24(4):5–15, April 1991.

[95] M. Kuga, K. Murakami, and S. Tomita. DSNS (Dynamically-hazard-resolved, Statically-code-scheduled, Nonuniform Superscalar): Yet another superscalar processor architecture. *ACM Computer Architecture News*, 19(4):14–29, June 1991.

[96] H. T. Kung. Why systolic architectures? *IEEE Computer*, 15:37–46, January 1982.

[97] S-Y. Kung, S. C. Lo, and P. S. Lewis. Timing analysis and design optimisation of VLSI data flow arrays. In *The Proceedings of the IEEE International Conference on Parallel Processing*, pages 600–607, 1986.

[98] C H. Lau. SELF: A self-timed systems design technique. *Electronics Letters*, 23(6):269–170, March 1987.

[99] L. Lavagno and A. Sangiovanni-Vincentelli. Automated synthesis of asynchronous interface circuits. In S. Furber and M. Edwards, editors, *The Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies*, Manchester, UK, March 1993. Elsevier Science Publishers.

[100] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17(1):6–22, January 1984.

[101] P. F. Lister and A. M. Alhelwani. Design methodology for self-timed VLSI systems. *IEE Proceedings-E Computer and Digital Techniques*, 132(1):25–32, January 1985.

[102] Å. Lunde. Empirical evaluation of some features of instruction set processor architectures. *Communications of the ACM*, 20(3):143–153, March 1977.

[103] T. Mano, F. Maruyama, K. Hayashi, T. Kakuda, N. Kawato, and T. Uehara. OCCAM to CMOS: An experimental logic design support system. In C. J. Koomen and T. Moto-oka, editors, *Computer Hardware Description Languages and their Applications: The Proceedings of the Decennial Caltech Conference on VLSI*, pages 381–390. North Holland, 1985.

[104] S. Manoharan and P. Thanisch. Assigning dependency graphs onto processor networks. *Parallel Computing*, 17(1):63–73, April 1991.

[105] R. M. Marshall. *Synthesis of Hardware Systems from Very High Level Behavioural Specifications*. PhD thesis, Department of Computer Science, University of Edinburgh, UK., December 1986. CST-41-86.

[106] A. J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. Technical Report Caltech-CR-TR-89-1, Department of Computer Science, California Institute of Technology, Pasadena, California, 1989.

[107] A. J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In W. J. Dally, editor, *The Proceedings of the 6th MIT Conference on Advanced Research in VLSI*, Cambridge, Mass., 1990. MIT Press.

[108] A. J. Martin. Asynchronous datapaths and the design of an asynchronous adder. Technical Report Caltech-CR-TR-91-08, Computer Science Department, California Institute of Technology, 1991.

[109] A. J. Martin. Tomorrow's digital hardware will be asynchronous and verified. In J. van Leeuwen, editor, *Algorithms, Software, Architecture: Proceedings of the IFIP Information Processing Conference*, pages 684–695. North-Holland, September 1992.

[110] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus. The design of an asynchronous microprocessor. In C. L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 351–373, Cambridge, Mass., 1989. MIT Press.

[111] A. McAuley. Four state asynchronous architectures. *IEEE Transactions on Computers*, C-41(2):129–142, February 1992.

[112] C. McCreary, A. A. Khan, J. Thompson, and M. E. McArdle. A comparison of heuristics for scheduling DAGs on multiprocessors. Technical Report CSE-93-07, Auburn University, Auburn, AL, 36849. USA., 1994.

[113] S. McFarlane and J. Hennessy. Reducing the cost of branches. In *The Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 396–403, June 1986.

[114] E. McLellan. The Alpha AXP architecture and 21064 processor. *IEEE Micro*, pages 36–47, June 1993.

[115] C. Mead and L. Conway. *Introduction to VLSI Systems*. Addison-Wesley, Reading, Mass., 1980.

[116] T. H.-Y. Meng, R. W. Brodersen, and D. G. Messerschmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer Aided Design*, 8(11):1185–1205, November 1989.

[117] R. E. Miller. *Switching Theory. Volume II: Sequential Circuits and Machines*. John Wiley and Sons, 1965.

[118] S. Mirapuri, M. Woodacre, and N. Vasseghi. The MIPS R4000 processor. *IEEE Micro*, pages 10–22, April 1992.

[119] C. E. Molnar, T-P. Fang, and F. U. Rosenberger. Synthesis of delay-insensitive modules. In Henry Fuchs, editor, *1985 Chapel Hill Conference on VLSI*, pages 67–86. Computer Science Press, 1985.

[120] S.-M. Moon and K. Ebcioglu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *The Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO'25)*, pages 55–71, 1992.

[121] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, pages 114–117, April 1965.

[122] D. Morris and R. N. Ibbett. *The MU5 Computer System*. The Macmillan Press, 1979.

[123] S. V. Morton, S. S. Appleton, and M. J. Liebelt. ECSTAC: A fast asynchronous microprocessor. In M. B. Josephs, editor, *The Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, pages 180–189, London, UK, May 1995. IEEE Computer Society Press.

[124] D. E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Vol. XXIX of The Annals of the Computation Laboratory of Harvard University*. Harvard University Press, 1959.

[125] D. E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *The Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.

[126] R. D. Mullins. A VLSI design methodology for asynchronous processor architectures. Technical report, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland, UK., May 1994.

[127] R. D. Mullins. An asynchronous superscalar RISC architecture. Master's thesis, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland, UK., September 1995.

[128] E. J. Muth. The production rate of a series of workstations with variable service times. *International Journal of Production Research*, 11(9):155–169, 1973.

[129] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura. TITAC: Design of a quasi-delay-insensitive microprocessor. *IEEE Design and Test of Computers*, pages 50–63, Summer 1994.

[130] A. Nicolau. Loop quantization or unwinding done right. In *The Proceedings of the 1st International Conference on Supercomputing*, pages 294–308, June 1987.

[131] C. D. Nielsen and A. J. Martin. A delay-insensitive multiply-accumulate unit. Technical Report CS-TR-92-03, Computer Science Department, California Institute of Technology, 1992.

[132] Y.-J. Oyang, C.-H. Wen, Y.-F. Chen, and S.-M. Lin. The effects of employing advanced branching mechanisms in superscalar architectures. *ACM Computer Architecture News*, 18(4):35–51, December 1990.

[133] K. V. Palem and B. B. Simons. Scheduling time-critical instructions on RISC machines. *ACM Transactions on Programming Languages and Systems*, 15(4):632–658, September 1993.

[134] C. H. Papadimitrou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal of Computing*, 19(2):322–328, April 1990.

[135] V. Patel and K. Steptoe. Evaluation of self-timed systems for VLSI. *Electronics Letters*, 25(3):215–217, February 1989.

[136] D. A. Patterson and C. H. Séquin. RISC I: A reduced instruction set VLSI computer. In *The Proceedings of the 8th International Symposium on Computer Architecture*, pages 443–457, May 1981.

[137] N. C. Paver. *The Design and Implementation of an Asynchronous Microprocessor*. PhD thesis, Department of Computer Science, University of Manchester, UK., 1994.

[138] G. Radin. The 801 minicomputer. In *The Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 39–47, March 1982.

[139] B. R. Rau and J. A. Fisher. Instruction-Level Parallel processing: History, overview and perspective. *The Journal of Supercomputing*, 7(1/2):9–50, May 1993.

[140] M. Rem. Concurrent computations and VLSI circuits. In M. Broy, editor, *Control Flow and Data Flow: Concepts of Distributed Programming*, pages 399–437. Springer-Verlag, 1986.

[141] M. Rem. Trace theory and systolic computations. In J. W. deBakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE: Parallel Architectures and Languages Europe*, volume 1, pages 14–34. Springer-Verlag, 1987.

[142] W. F. Richardson. *Architectural Considerations in a Self-Timed Processor Design*. PhD thesis, Department of Computer Science, University of Utah, UT, USA., February 1996. CSTD-96-001.

[143] W. F. Richardson and E. L. Brunvand. The NSR processor prototype. Technical Report UUCS-92-029, Department of Computer Science, University of Utah, USA., 1992.

[144] M. Roncken. Partial scan test for asynchronous circuits illustrated on DCC error corrector. In *The Proceedings of the International Symposium on Advanced Research on Asynchronous Circuits and Systems (ASYNC'94)*, Salt Lake City, Utah, USA, March 1994. IEEE Computer Society Press.

[145] M. Roncken and R. W. Saeijs. Linear test times for delay-insensitive circuits: A compilation strategy. In S. Furber and M. Edwards, editors, *The Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies*, Manchester, UK, March 1993. Elsevier Science Publishers.

[146] O. Salomon and H. Klar. Self-timed fully pipelined multipliers. In S. Furber and M. Edwards, editors, *The Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies*, Manchester, UK, March 1993. Elsevier Science Publishers.

[147] K. C. Saraswat and F. Mohammadi. Effect of scaling of interconnections on the time delay of VLSI circuits. *IEEE Journal on Solid-State Circuits*, SC-17(2):275–280, April 1982.

[148] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. The MIT Press, 1989.

[149] F. Schalij. The Tangram manual. Technical Report Technical Report UR 008/93, Philips Research Labs Eindhoven, 1993.

[150] C. L. Seitz. System Timing. In C. Mead and L. Conway, editors, *Introduction to VLSI Systems*, chapter 7, pages 218–262. Addison-Wesley, 1980.

[151] R. Sethi. Scheduling graphs on two processors. *SIAM Journal of Computing*, 5(1):73–82, 1976.

[152] A. Severson and B. Nelson. Throughput in a Counterflow pipeline processor. *ACM Computer Architecture News*, 23(1):5–12, March 1995.

[153] J. E. Smith. A study of branch prediction strategies. In *The Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, May 1981.

[154] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, May 1988.

[155] S. P. Song, M. Denman, and J. Chang. The PowerPC 604 RISC microprocessor. *IEEE Micro*, 14(5):8–17, October 1994.

[156] J. Sparsø, C. D. Neilsen, L. S. Nielsen, and J. Staunstrup. Design of self-timed multipliers: A comparison. In S. Furber and M. Edwards, editors, *The Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies*, Manchester, UK, March 1993. Elsevier Science Publishers.

[157] R. F. Sproull, I. E. Sutherland, and C. E. Molnar. Counterflow pipeline processor architecture. Technical Report SMLI TR-94-25, Sun Microsystems Laboratories Inc., April 1994.

[158] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[159] S. M. Sze. *VLSI Technology*. McGraw Hill, 1983.

[160] Y. K. Tan and Y. C. Yim. Self-timed system design technique. *Electronic Letters*, 26(5):284–286, 1990.

[161] G. Theodoropoulos. *Strategies for the Modelling and Simulation of Asynchronous Computer Architectures*. PhD thesis, Department of Computer Science, University of Manchester, UK., September 1995.

[162] J. E. Thornton. *Design of a Computer: The Control Data 6600*. Scott Foresman and Company, 1970.

[163] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[164] I. P. Tzonos. A VLSI library of asynchronous cells. Master's thesis, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland, UK., September 1995.

[165] J. T. Udding. *Classification and Composition of Delay-Insensitive Circuits*. PhD thesis, Eindhoven University of Technology, September 1984.

[166] J. T. Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing*, 1:197–204, 1986.

[167] N. Ullah and M. Holle. The MC88110 implementation of precise exceptions in a superscalar architecture. *ACM Computer Architecture News*, 21(1):15–25, March 1993.

[168] J. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10:384–393, 1975.

[169] S. H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[170] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij. A fully asynchronous low-power error corrector for the DCC player. *IEEE Journal of Solid State Circuits*, 29(6):1429–14398, 1994.

[171] K. van Berkel, J. Kessels, M. Roncken, R. W. Saeijs, and F. Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *The Proceedings of the European Design Automation Conference*, pages 384–389, 1991.

[172] J. L. A. van de Snepscheut. *Trace Theory and VLSI design*, volume 200 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[173] R. van de Wiel. High-level test evaluation of asynchronous circuits. In M. B. Josephs, editor, *The Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, pages 63–71, London, UK, May 1995. IEEE Computer Society Press.

[174] T. Verhoeff. Delay-insensitive codes - An overview. *Distributed Computing*, 3:1–8, 1988.

[175] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison-Wesley, Reading, Mass., 1985.

[176] W. A. Wulf. Compilers and computer architecture. *IEEE Computer*, 14:41–47, July 1981.

[177] M. Yoeli. Structured design of the control parts of self-timed VLSI systems. In O. N. Garcia and X. Zhang, editors, *The Proceedings of 2nd International Conference on Computer and Applications*, pages 839–841. IEEE Computer Society Press, 1987.

[178] M. Yoeli. Net based synthesis of delay-insensitive circuits. Technical Report 609, Department of Computer Science, Technion - Israel Institute of Technology, Haifa, Israel, February 1990.

[179] M.-L. Yu and P. A. Subrahmanyan. Hazard-free asynchronous circuit synthesis. In S. Furber and M. Edwards, editors, *The Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies*, Manchester, UK, March 1993. Elsevier Science Publishers.

[180] J. Yuan and C. Svensson. High-speed CMOS circuit techniques. *IEEE Journal on Solid-State Circuits*, SC-24(1):62–70, February 1989.