Dynamic instruction scheduling and data forwarding in asynchronous superscalar processors

Robert D. Mullins

Doctor of Philosophy
University of Edinburgh
2001

Abstract

Improvements in semiconductor technology have supported an exponential growth in microprocessor performance for many years. The ability to continue on this trend throughout the current decade poses serious challenges as feature sizes enter the deep sub-micron range. Problems due to increasing power consumption, clock distribution and the growing complexity of both design and verification, may soon limit the extent to which the underlying technological advances may be exploited. One approach which may ease these problems is the adoption of an asynchronous design style – one in which the global clock signal is omitted. Commonly-cited advantages include: the ability to exploit local variations in processing speed, the absence of a clock signal and its distribution network, and the ease of reuse and composability provided through the use of delay-insensitive module interfaces. While the techniques to design such circuits have matured over the past decade, studies of the impact of asynchrony on processor architectures have been less common. One challenge in particular is to develop multiple-issue architectures that are able to fully exploit asynchronous operation. Multiple-issue architectures have traditionally exploited the determinism and predictability ensured by synchronous operation. Unfortunately, this limits the effectiveness of the architecture when the clock is removed. The work presented in this dissertation describes in detail the problems of exploiting asynchrony in the design of superscalar processors. A number of techniques are presented for implementing both data forwarding and dynamic scheduling mechanisms, techniques that are central to exploiting instruction-level parallelism and achieving high-performance. A technique called instruction compounding is introduced, which appends dependency information to instructions during compilation, which can be exploited at run-time. This simplifies the implementation of both the dynamic scheduling and data-forwarding mechanisms. The performance characteristics of the different techniques are compared through simulation. Results show that an asynchronous version of a generic synchronous superscalar processor can provide similar performances. Although the performances of instruction compounded and queue-based asynchronous implementations are lower, their designs are far less complex, with scope for better performance with improved compiler support.

Acknowledgements

I would like to thank Damal Arvind for his support and guidance throughout my PhD. I would also like to thank everyone in the department who offered advice, encouragement and friendship, especially Christos, Dominic, John, Lawrence and Salvador. I would also like to thank George, Paul, Peter, and Simon, at Cambridge for their support and advice during the last twelve months. Many special thanks to Arvind, George, Simon and Tania for spending time proofreading the dissertation and providing invaluable feedback. Finally, I would like to thank Tania, Mum and Dad, for their continual support and patience!

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

Some of the work presented in this thesis has already been published in:

ARVIND, D. K., AND MULLINS, R. D. Instruction issue and data forwarding mechanisms for asynchronous superscalar processors. In *Proceedings of the Workshop on Complexity-Effective Design (WCED'00, held in conjuction with ISCA-27)* (Vancouver, B.C., June 2000).

ARVIND, D. K., AND MULLINS, R. D. A fully asynchronous superscalar processor. In *International Conference on Parallel Architectures and Compilation Techniques*, Newport Beach, California, October 1999.

ARVIND, D. K., AND MULLINS, R. D. Static and Dynamic Instruction Compounding in an Asynchronous Superscalar Architecture. In *Proceedings of the 6th UK Asynchronous Forum* (Manchester, UK, July 1999).

ARVIND, D. K., AND MULLINS, R. D. Redefining the SW/HW boundary in Asynchronous ILP Architectures In *Proceedings of the 3rd UK Asynchronous Forum* (Edinburgh, Scotland, Dec. 1997).

ARVIND, D. K., AND MULLINS, R. D. Instruction compounding. In *Proceedings of the 1st UK Asynchronous Forum* (Edinburgh, Scotland, Dec. 1996).

(Robert Mullins)

Table of Contents

Chapter 1

Introduction

Advances in semiconductor fabrication, processor architecture, and compiler technology have enabled an exponential growth in microprocessor performance since the early 1970s. The ability to sustain such rates of growth in this century depends upon both the continued scaling of CMOS processes and the existence of architectures and design techniques which are able to exploit them. Studies suggest that improvements in process technologies will continue for at least another ten years [1]. However, the designers of high-performance processors already face serious problems in fully exploiting deep sub-micron technologies.

Higher performance has historically been achieved through a combination of increased levels of instruction-level parallelism (ILP) and higher clock frequencies. The introduction of deeper pipelines and advances in circuit design techniques have allowed clock frequencies to increase faster than that made possible by reductions in gate delay alone. The ability to maintain die sizes while reducing feature sizes has also provided large increases in transistor budgets. This has allowed ILP to be boosted through scope for greater speculation, dynamic scheduling, and the duplication of functional units. Architecturally speaking, general purpose high-performance processor design has converged, with the vast majority of implementations adopting a superscalar RISC organisation to achieve both a high clock frequency and to efficiently exploit the available ILP.

However increases in complexity and transistor counts cannot be achieved without additional costs. Increases in clock frequencies and ILP, and greater transistor counts have resulted in a sharp growth in power consumption. Techniques for reducing and managing power have become important factors in the design of all high-performance VLSI systems due to the problems of both supplying power and the costs associated with cooling. Many of the current low-power techniques are focussed on reducing the power dissipated when transistors switch (dynamic power); these include: clock and signal gating, the use of multiple on-chip supply voltages, the dynamic scaling of supply voltage, and more recently dynamic reconfiguration of components to minimise load capacitances [13, 25, 43]. Additional techniques will also be required to handle the predicted increases in power due to leakage currents (static power), which are increasing by around a factor of five per process generation [19]. Techniques proposed to combat static power usage include: support for multiple threshold voltages, minimising the use of wide transistors, and the use of power gating. Projected increases in both static and dynamic power suggest that designs will be increasingly influenced by power considerations. In architectural terms, designs will become increasingly irregular in order to fully realise potential power savings. Limited power budgets will see the creation of designs which carefully allocate the number of components running at the highest frequency and supply voltage. One vision of such a processor, describes a high frequency core supported by lower frequency "helper engines". In the initial design process functionality that may be implemented at a low frequency, for example as a result of the availability of higher levels of parallelism, is displaced to the helper engines to minimise overall power consumption [128].

A second technology trend is the increased cost of communication. While local interconnect delays almost scale with gate delays, global interconnect does not. This has led to an increasing proportion of the clock period being consumed by interconnect delay. In the past, significant communication latencies have only been encountered when communicating off-chip or at the board level. Future chip designers will need to operate in an environment where a communication spanning the width of the chip will take many clock cycles. This problem has recently been illustrated in the design of the Pentium IV processor where specific pipeline stages are provided solely for the purpose of communication. Architectural techniques such as the clustering of functional units have also been necessary due to increasing interconnect delays [75]. The increasing interconnect delays relative to the delay of transistors will lead to an increase in the range of on-chip delays. Predicting the precise time required for such communications will also become increasingly difficult as technology scales. Such calculations require a complex analysis of capacitances and consideration of increasing process and environmental variations [167]. Even with improvements in interconnect materials, such as the introduction of copper, and dielectric insulators, interconnect delay will still have a significant impact on the development of future architectures.

The trends described above are also expected to be accompanied by a move away from the implementation of a single clock domain. Local clock frequencies are predicted to rise above 10Ghz, making it both unrealistic and undesirable to maintain global synchronisation. At present multiple clock frequencies are usually only required to support off-chip accesses. In the future the number of individual clock domains and the associated synchronisation problems are likely to rise considerably. Even with a move towards multiple clock domains, the generation and distribution of clocks with cycle times of much less than a nanosecond will be problematic. Significant resources are already required to analyse and construct clock distribution networks [34, 40]. This task is complicated by the introduction of low-power techniques such as clock gating, which can potentially introduce additional skew [145].

The performance of synchronous designs will of course continue to grow even in the presence of such technology trends. Engineering solutions will inevitably be found and applied. The cost of such an approach will be a sharp rise in the complexity of both the design and verification processes. However, as design team sizes cannot grow indefinitely the degree to which the underlying technology is exploited will be reduced. The fundamental problem is that the design environment is steadily moving away from one in which a synchronous timing regime may be applied efficiently. Traditionally, the clock has been viewed as a simplifying assumption providing opportunities to exploit predictable behaviour and minimise critical paths dominated by gate delays. Future designs will pose different problems requiring the creation of far more heterogeneous systems to tackle power and communication issues. Another key requirement in applying a synchronous approach efficiently is the ability to accurately predict on-chip delays. Future Deep Sub-Micron (DSM) technologies will provide a number of challenges in this area, these include: the data-dependent nature of interconnect delays due to coupling effects, state-dependent timing effects in newer technologies such as Silicon-On-Insulator (SOI), voltage drops due to large cycle-to-cycle current swings, and increased thermal gradients [1]. As synchronous clock frequencies must be set by considering worst-case delays, difficulties in predicting delays and an increase in data-dependent delays will mean that on average the amount of useful work performed in a clock cycle will drop.

1.1 A clock-free design approach

Shifting design trade-offs have led to the proposal that future VLSI systems should be designed to operate asynchronously – removing the clock completely and freeing the system from lockstep operation [124, 140, 59, 149]. In contrast to synchronous systems, asynchronous designs operate in an event-driven manner.

Operations are initiated with local communications and completion is detected explicitly. In fact, the correct operation of the circuit is in many cases independent of the delays of both gates and interconnect. Another consequence of the way in which control is implemented is that superfluous switching activity is often minimised. This can be considered as equivalent to a rigorous clock and signal gating scheme, but unlike a synchronous design this behaviour is implicit in the construction of many asynchronous systems. As operations are now initiated in a data-driven fashion, free from any global timing constraints, switching activity is also more evenly distributed. This helps to reduce the problems of high cycle-to-cycle current variations and the clock-related electromagnetic emissions found in synchronous designs. Finally, asynchronous techniques allow the construction of circuits which operate correctly regardless of actual circuit delays. While this reduces the need for precise timing information from a verification perspective, it also has the potential to improve performance by exposing and exploiting actual logic and interconnect delays.

Asynchronous or self-timed techniques are already in use in synchronous processors where they are designed to operate within the constraints of the synchronous clock period [78]. They are also generally accepted as one possible solution to the problem of providing a communication structure between many different synchronous clock domains [1]. This has led to the development of Globally-Asynchronous Locally-Synchronous (GALS) approaches [28, 164, 20, 93]. The use of fully asynchronous techniques is far less widespread and is commercially limited to a few designs with low power or EMI requirements [149]. One reason for this is that in the past asynchronous design techniques have represented both area and performance overheads due to the additional logic required for communication and completion detection. With a significantly different set of design trade-offs currently facing designers the relative cost of each approach to system timing is set to change. In the long term, the ability of an asynchronous approach to support an increasingly heterogeneous timing environment and to minimise power consumption will become increasingly important. The reduced importance of absolute timing information and the composability offered by delay-insensitive module interfaces also aids in managing verification and design complexity. Together, these trends are set to make a fully asynchronous approach increasingly attractive. At the same time, retaining a synchronous design style will lead to increasingly inefficient implementations.

1.2 Aims of the thesis

One obstacle to fully exploiting an asynchronous approach is the development of architectures that are able to operate efficiently in an asynchronous environment. While the synthesis and verification of asynchronous circuits has matured in the last ten years [60], the design of high-performance asynchronous processors has received less attention.

Many mechanisms in existing superscalar designs exploit the deterministic and predictable nature of a synchronous system. These architectural features may only be retained in an asynchronous design through the introduction of additional synchronising communications. These synchronisations force the asynchronous system to operate in a pseudo-synchronous manner. High-level synchronisations potentially expose worst-case delays and limit the extent to which actual delays may be exploited and the control overheads hidden.

Two mechanisms that traditionally rely on global synchronisation are data forwarding and dynamic scheduling. These operations typically require communication between different pipeline stages that introduces the type of synchronisations outlined above.

The aim of this dissertation is to investigate the impact of asynchrony on the architecture of superscalar processors. Initially the transformation of traditional synchronous architectures is explored. A number of techniques are presented which can be used to provide mechanisms for out-of-order dispatch and data-forwarding suitable for an asynchronous implementation. Later designs extend these ideas by exploiting additional dependency information provided to the architecture by the compiler. Each of the proposed designs is described and their performance compared with a generic synchronous superscalar processor through simulations.

1.3 Thesis Outline

A summary of the remaining chapters is given below.

Chapter 2 details four main background areas: (1) system timing, contrasting the synchronous and asynchronous timing regimes; (2) the impact of scaling CMOS technologies; (3) asynchronous processor architecture, presenting an overview of previous work in this area and finally (4) an overview of the architecture of existing synchronous multiple-issue machines.

Chapter 3 introduces the operation of a generic superscalar processor. The translation of this synchronous style architecture into one suitable for an asynchronous implementation is then investigated. Alternative schemes for both data forwarding and dynamic scheduling are presented which are able to better exploit an asynchronous implementation.

Chapter 4 explores how dependency information extracted at compile-time may be used to simplify the datapath. The technique of *instruction compounding* is introduced which exploits information appended to instructions at compile-time to provide a simplified implementation of both dynamic scheduling and data-forwarding mechanisms.

Chapter 5 explores the performance characteristics of each of the architectures presented in Chapters 3 and 4. A fully synchronous architecture is also simulated and compared.

Chapter 6 summarises the work presented and discusses future work, including the development of appropriate formal verification techniques and compiler technology. Promising future directions are also identified and overall conclusions presented.

Chapter 2

Background

2.1 Introduction

This chapter explores the characteristics of both synchronous and asynchronous design styles; trends in process technology are used to highlight the limitations of enforcing global synchronisation with the growth in design sizes and the increase in power requirements; an overview of the state-of-the-art in asynchronous processors and contemporary synchronous superscalar architectures is also provided.

2.2 The Timing Regime

The physical nature of the components of a VLSI system make it impossible to compose them without considering a suitable method for orchestrating their operation. In contrast to systems where implicit synchronisation exists, the transistors which form the building blocks of a VLSI system cannot be guaranteed to take an exact time to perform an operation. As a result, two distinct approaches exist for designing VLSI systems. The first is to ensure that when logic gates are composed they behave in a predictable and predefined manner (asynchrony). The second is to accept any behaviour from the system as long as it settles to provide a stable output within a given fixed time interval (synchrony). The following two sections provide a brief overview of the synchronous and asynchronous approaches.

2.3 Synchronous Design

The vast majority of digital circuits designed today operate synchronously. The aim of such an approach is to create a system whose observable behaviour develops in a predictable and timely fashion. In a synchronous system this is achieved by organising circuit activity into discrete bursts followed by periods where the system is guaranteed to be quiescent. A new burst of activity is then initiated as soon as the results from the previous one have been saved. The system progresses in lockstep in this way under the control of a globally distributed timing reference or clock signal. The clock period is set to represent the slowest operation which will ever need to be performed. This guarantees that at the end of each clock cycle all operations will have been completed.

The design of synchronous systems begins at the cycle level, which makes it easy to predict performance for a given clock frequency. Global synchronisation provides the designer with complete control over system behaviour at this level. Operations may be scheduled during the design process to take place at a particular time, or within a given clock cycle. From an architectural perspective, the design may be optimised with knowledge of when a future operation will be performed and with the availability of global state information. This often allows control circuits to run in parallel with the datapath, preparing control signals for the next cycle while datapath computations are performed. Conceptually the circuit design process is simplified by hiding the transient behaviour of the circuits between clock edges. This allows logic to be optimised to produce results as quickly as possible, regardless of the intermediate values which may be generated.

As the clock controls the rate at which operations are performed in such a system, increasing the clock frequency will always improve the performance of a particular design. This often means that the primary design goal is to minimise the critical path until the required clock frequency and performance is met. The probability of taking a particular critical part is unimportant, only that it represents the worst-case delay. Optimisations at the circuit level are simplified by the fact that the logic found between clocked registers is purely combinational. The need to minimise only the worst-case logic path has a significant impact on the structure of the final circuit. In practice, high clock frequencies are achieved both through circuit level optimisations and architectural innovations. Pipelining (see Section 2.6.1) in particular has played a key role in achieving high clock frequencies in modern processors. In addition, the simplicity and effectiveness of pipelining schemes means that it often provides significant reductions in the energy-delay product [54].

As resources in a synchronous system are usually allocated at the cycle level, resource utilisation is also often measured in terms of clock cycles. This is in some ways misleading as the real resource utilisation within a clock period is in fact much lower than 100%. The clock period not only accounts for the system's critical path, but must also include register setup and clock-to-output delays. Delay modeling inaccuracies and environmental and process variations are also accounted for with the inclusion of a clock safety margin. Finally, imperfections in the clock generation (jitter) and distribution (clock skew) also require clock periods to be extended. The key requirement in order to exploit the implementation technology when using a synchronous design style is the early and accurate prediction of absolute delays in the design flow. This allows the identification of critical paths and permits the necessary modifications to be made at the circuit and architectural level.

A final issue with respect to synchronous circuits are the problems associated with *metastability* [27]. Metastability may occur when flip-flop setup times are violated, for example when attempting to synchronise an asynchronous input. If this occurs the output of the flip-flop may hang at an intermediate voltage while a decision to either rise or fall to a true logic level is made. To provide additional time for metastability to resolve, such inputs are usually synchronised by passing them through a number of flip-flops (usually two). Although the probability of failure may be practically removed, the use of additional flip-flops increases latency. This type of synchronisation is also required when interfacing two or more independently clocked synchronous modules.

2.4 Asynchronous Design

Systems implemented using an asynchronous design approach operate without reference to a global clock signal. In the absence of global synchronisation, communication of data and the sequencing of control actions must be implemented explicitly. In the case of communication this is made possible through the use of handshaking and other delay-insensitive signalling protocols. Control circuits to implement such functionality are distributed throughout the system wherever communication or other control tasks must be performed. In such a system, correct operation is no longer dependent on meeting strict timing requirements. Instead, it relies only on controlling the *order* in which events are permitted to occur. In fact, it is often the case that due to the data-dependent nature of delays the precise ordering of independent events within the system will be non-deterministic. The following sections provide a brief overview of the implications of taking such an approach for both the circuit design and the architecture.

2.4.1 Asynchronous circuits

The value on a wire in a digital system changes whenever a transition from one logic level to another occurs. Within an asynchronous circuit this is often called an *event.* Events which occur on non-clock signals in synchronous systems are unable to influence functionality, only their value when an event occurs on the clock wire is important. In practice, due to varying delays through different logic paths, a particular wire in a block of combinational logic may perform a number of transitions before reaching a stable value. The potential for the generation of such intermediate values on the outputs of logic gates is called a *hazard.* Hazards may also result in incomplete or non-monotonic transitions. In an asynchronous system, where all events potentially influence behaviour, it is important to ensure that such glitches do not cause the circuit to malfunction.

Due to the potential for hazards, the number of useful asynchronous circuits that operate correctly free from any assumptions about wire and gate delay is limited. Practical synthesis and design methodologies usually operate under a more relaxed timing model, *e.g. speed-independence*, where the delay of wires is considered to be insignificant. Such assumptions are realistic when they are applied locally, to so called *equipotential regions.* Communication between these regions may be handled using delay-insensitive signalling conventions. *Self-timed systems*, introduced by Seitz in [124], is an example of a scheme which follows such an approach.

In general, the process of designing asynchronous circuits is simplified by local timing assumptions about the environment and the internal delays of the circuit. Over time an increasing number of internal timing assumptions have been introduced in order to optimise designs. Possible delay models range from the fully *Delay-Insensitive* (DI), where no constraint is placed on any gate or wire delay, to *timed-circuits* [65, 100, 150] where bounded-delay assumptions are made about both internal delays and the environment. More recently, relative timing assumptions [138] have been used to optimise the synthesis of asynchronous circuits. A more complete introduction to the theory of asynchronous circuits may be found in [60].

The complexities of ensuring that asynchronous circuits behave correctly under a particular delay model often preclude the use of *ad-hoc* circuit design methods, or the reliance on simulation alone to ensure correct operation. The need to specify concurrency and handle delay assumptions at the circuit level also means that traditional state-machine based specifications cannot always provide a suitable formalism for synthesis or verification. As a result, much of the research in asynchronous systems has focused on techniques for specification, synthesis and formal verification. Synthesis techniques broadly fall into two categories: graph based synthesis techniques and syntax-directed translation approaches. Graph-based methods manipulate Petri-net or signal transition graph (STG) specifications. Such representations offer a simple way to describe concurrency, this is important as many asynchronous control circuits have a high-degree of concurrency, permitting a complex interweaving of events at their interface. A number of freely available CAD packages may be used to automate the tasks of state assignment and technology mapping – these include Petrify [31] and SIS [80]. Such approaches are usually limited to specifying small asynchronous control circuits due to the problems of state space explosion. An alternative is to specify systems using parallel programming languages that are then translated to asynchronous circuit implementations. The translation process compiles high-level descriptions into low-level programs whose operations map directly to a library of asynchronous components. The final circuit may be improved further by applying a range of peep-hole optimisations. Examples of such systems include: work on translating Occam to delay-insensitive circuits [23], the Philips Tangram system [17], Manchester's Balsa system [14] and the CSP-based approach taken at Caltech [86]. In general, direct translation methods provide a guaranteed route to an implementation, regardless of the complexity of the specification. In contrast, state-based techniques are limited to smaller specifications but provide the ability to create highly optimised solutions. Attempts to combine both approaches have also been explored [111, 77].

The absence of a clock signal in an asynchronous system poses the question of how we can choose between two potential inputs without suffering from the problems of metastability. While it is impossible to detect or remove metastability, asynchronous circuits are free to wait for it to resolve. This is in contrast to a synchronous circuit where the hard deadline for metastability resolving is the arrival of the next clock edge. Circuits that implement mutual-exclusion or arbitration functions are provided with a metastability filter, preventing metastable states propagating to their outputs [124, 123]. Multi-way arbitration is also possible by combining two or three-way arbiters into tree structures [69, 159] or through the use of token based ring arbiters [157].

2.4.2 Communication and delay-insensitive signalling

In a clocked system all communication takes place simultaneously at the end of each clock cycle. In the absence of global synchronisation, asynchronous systems must initiate communication explicitly. The *handshake* protocol provides a suitable delay-insensitive signalling convention for such communications. A *request* signal is used to indicate the desire of the sender to communicate, while an *acknowledgement* signal produced by the destination indicates that the communication has completed.

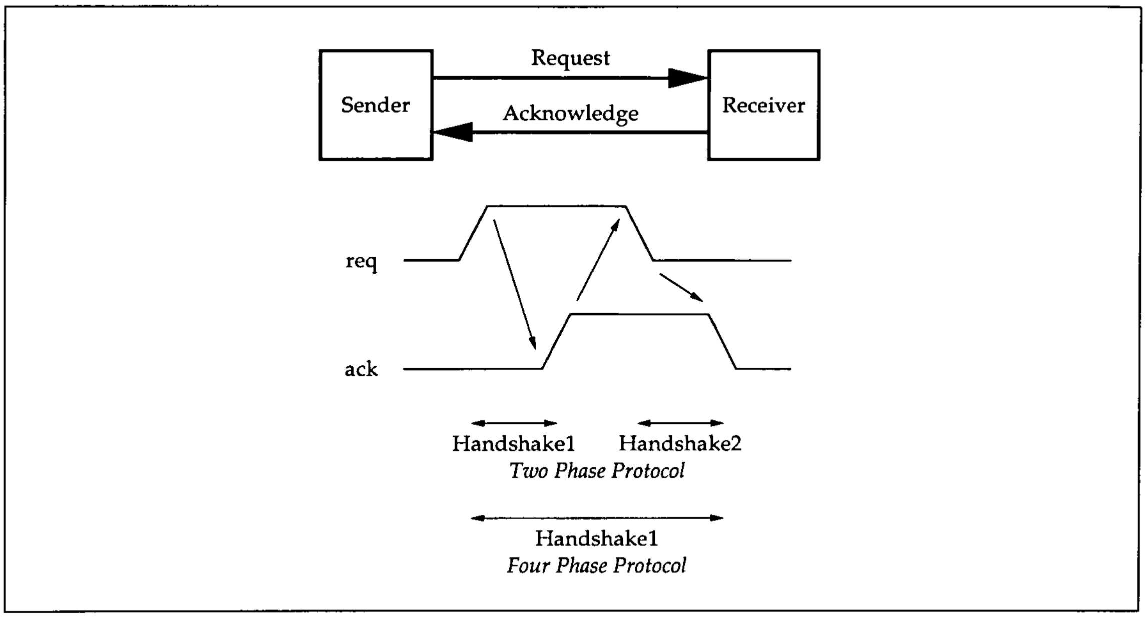


Figure 2.1: Two and Four phase handshaking protocols

Figure 2.1 illustrates two such protocols. These protocols differ in the number of signal transitions used to complete a communication. In the *two-phase* or *transition signalling* scheme, a single event is used to generate a request prompting a single event as an acknowledge. As a consequence, the levels on the acknowledge and request wires have no meaning. An alternative is to use a *four-phase* or *level-signalling* protocol and ensure both request and acknowledge wires are returned-to-zero (RTZ) after each communication. In this case the levels on the request and acknowledge wires indicate a particular phase of the handshake.

The choice of handshaking protocol influences the design of the corresponding interface circuits. In general, it is accepted that four-phase protocols result in smaller interface implementations, while the fewer transitions required by a two-phase approach may result in power and performance advantages. The trade-off is complicated by architectures that effectively hide the RTZ phase of the four-phase handshake by performing other useful operations in parallel. Timing assumptions can also be introduced into the protocol to improve performance [90]. Four-phase protocols may also be competitive in terms of power consumption when simplifications to the interface circuitry due to a reduction in state are considered. An alternative to using a transition to represent an event is to use a complete pulse, such ideas have recently been revisited in [112, 139]. This approach potentially combines the simplicity of the two phase protocol with the advantages of returning-to-zero as found in the four phase system.

In addition to performing purely control related tasks, the handshaking protocols described above are also used to transmit data. As the simultaneous arrival of both data and request signals cannot be guaranteed under a delay-insensitive delay model, a technique is required to detect when data is present. The arrival of a particular bit of data is only possible if it produces an event on a wire. If the data was sent unencoded this would mean that only those bits whose values changed could be detected. To overcome this limitation a number delay-insensitive codes have been devised [63, 152, 21]. Dual-rail encoding is one such technique. Here two bits are used to encode each bit of data, allowing the presence or absence of valid data to be established. The encoding is illustrated in Figure 2.2, where $D0$ and $D1$ are the two wires used to transmit the data. A high logic level on both wires is not usually permitted in normal operation and may be used to indicate an error has occurred.

Data	D0	D1
No value	0	0
Logic 0	1	0
Logic 1	0	1
Illegal	1	1

Figure 2.2: Dual-rail delay insensitive encoding scheme

One potential disadvantage of such techniques is the circuit and performance overhead required to encode and decode data. An alternative to detecting the presence of data is to simply introduce a safety margin by delaying the request event. This ensures that the request arrives after new data values have been established at the receiver's interface. This approach is called *bundled data.*

Both the bundled-data and delay-insensitive encoding schemes are also used when completion signals must be generated for combinational logic, *e.g.* a functional unit. In such cases, both schemes may be employed together. For example, delay-insensitive codes may be preferred for paths with significant data-dependent delay while other delays are modelled with a bundled data approach [51]. Bit-sliced and pipelined completion detection techniques have also been developed in order to achieve high performance [88].

Alternatively, the inactivity at the end of a computation may be detected directly at the circuit level, either through current sensing techniques [32, 33] or by monitoring the activity of nodes within the combinational logic [57]. The bundled data approach may also be extended to model a range of delays. Techniques exist for dynamically selecting a fixed delay depending on input data values to avoid worst-case performance [104].

2.4.3 Control circuit architecture

Control in synchronous systems tends to be centralised. This is a consequence of the availability of a global system state, which is readily available due to the lockstep operation of both control circuits and the datapath. In contrast, the interface between control circuits and the datapath in an asynchronous environment is an event-driven one. The need to service local communications between different control and datapath circuits efficiently suggests that control should be distributed.

A fine-grain highly concurrent control structure is required to produce the complex interweaving of events necessary to exploit actual circuit delays. An excellent example of how distributed control may be utilised is the *micropipeline* [140]. Figure 2.3 shows two pipelines (with processing logic omitted), the uppermost is a traditional synchronous pipelined constructed from edge-triggered flip-flops, while the micropipeline version is shown below. The only control signal required in the synchronous case is a simple square-wave clock signal. Each time a rising clock edge is received by the flip-flop's clock input, data in the pipeline is shifted forward by one stage.

The asynchronous version has no global control signals, each stage has a DI interface, which is used to communicate with the previous and next stages. Each stage uses these local handshake signals to determine the earliest time at which the next stage may receive its data. The example shown here uses a two-phase handshaking protocol as described in the previous section. Only a single gate is required to produce the necessary control for each stage. The *Muller C-elements* [99] used to produce the necessary handshaking signals may be thought of as an AND-gate for events. Only when an event has occurred on both inputs will an event occur on the output. The C-element's truth table and a possible standard cell implementation are shown in Figure 2.4. Fixed delays account for the logic and interconnect between each pipeline stage (bundled-data), although completion detection techniques may also be used. The state storing elements are also modified in the asynchronous case to operate under the control of events.

The resulting behaviour creates an elastic pipeline where both the number of

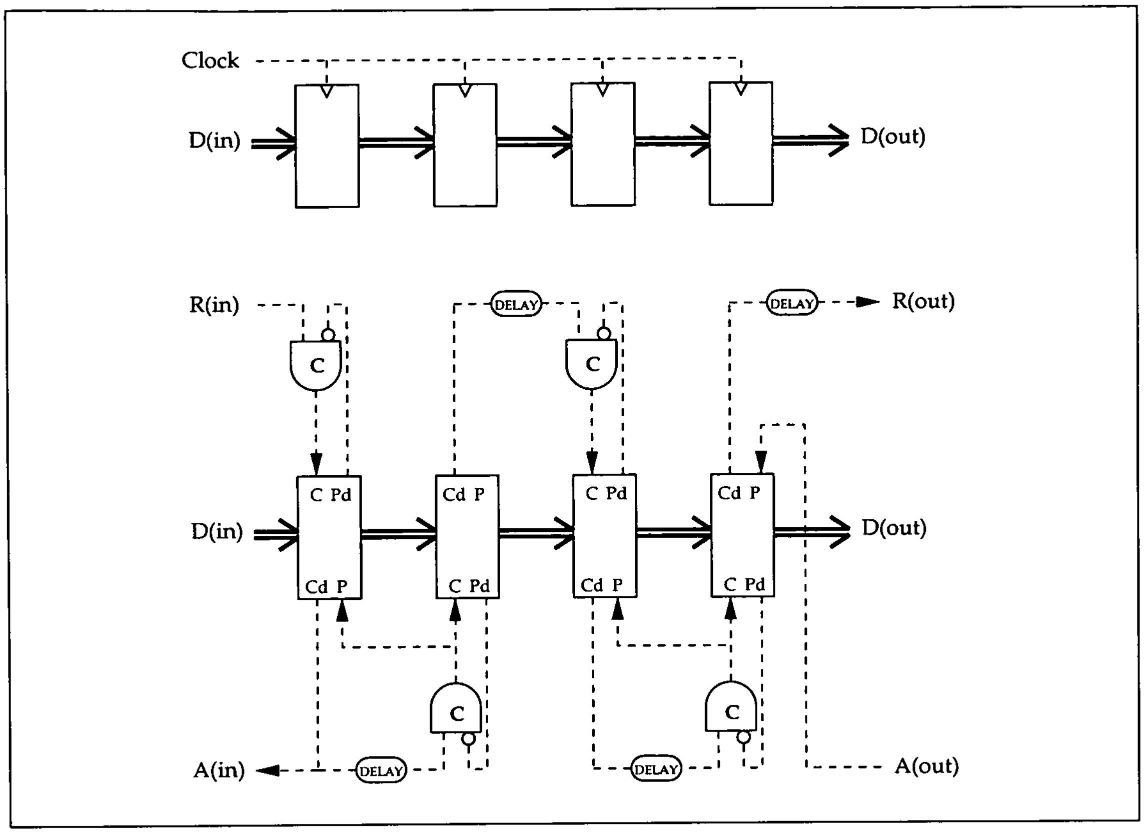


Figure 2.3: Synchronous and Asynchronous Pipelines

data items and the rate at which they move through different stages may vary. Such structures are potentially able to exploit variations in the delay of each stage. With adequate buffering and completion detection this may allow performance to tend towards the average case as opposed to the worst [73]. Two-dimensional micropipelines [55] and micronets [9, 8, 117] have also been explored. The design of the micropipeline latch control logic has received much attention, and many designs have been suggested for both two and four-phase styles [3, 44, 141, 165].

The organisation of control is critical to obtaining high-performance in an asynchronous environment. The implementation of control in a synchronous system is simplified by the fact that the granularity of parallel operations is in most

X	Y	Z_{n+1}
0	0	0
0	1	Z_n
1	0	Z_n
1	1	1

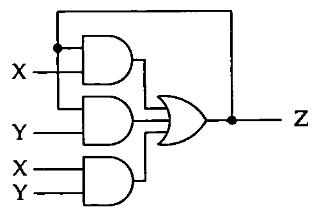


Figure 2.4: Truth table for C-element and a standard cell implementation

cases limited to the cycle level. In many cases control is able to run in parallel with datapath operations due to the deterministic and predictable nature of the system. Asynchronous designs must expose fine-grain parallelism in order to exploit local timing variations, but must also ensure that the handshaking and other control signals themselves do not represent a significant performance overhead. These aims influence the design of both circuits and architecture. Trade-offs are often necessary between parallelism, circuit complexity and the resulting speed of circuits. These problems, common to the designers of synchronous systems, must now be made at a much lower level of design.

A number of common problems which may limit the performance of asynchronous designs are enumerated below. These issues impact on both the design of local control circuits and overall architecture.

1. The generation of handshaking signals in order to communicate between control or datapath components represents a control overhead.

2. Overheads exist due to completion detection logic or delay matching safety margins.

3. A computation's average delay may naturally tend towards its worst-case, providing little scope for exploiting local variations in delay.

4. Buffering will often be required to exploit variations in datapath delays, unfortunately the buffer itself will introduce an additional delay.

5. System performance may be limited by bottlenecks that are unaffected by the choice of timing regime, such as the latency associated with accessing main memory.

6. Synchronisations may be required that mitigate the advantages of exposing a range of processing or communication delays.

7. When multi-way arbitration is required in an asynchronous system it may be slow and complex. Equivalent operations are often simplified in the presence of global synchronisation.

8. The ability to optimise asynchronous logic may be restricted by the need to consider its transient behaviour.

One final consideration is the problem of ensuring that correctness is maintained at the architectural level. While distributing control may produce the necessary concurrency, care must be taken to avoid potential deadlock conditions.

As a result formal verification is often required to ensure that such situations cannot occur.

2.4.4 Mixing Synchronous and Asynchronous Techniques

Globally Asynchronous Locally Synchronous (GALS) systems [28, 164, 61, 120, 6] aim to exploit many of the advantages of both clocked and asynchronous design styles. Synchronous modules, potentially working at a range of clock frequencies, are composed using an asynchronous communication structure. This allows synchronous design techniques to be applied locally, where interconnection delays are relatively small and global synchronisation may be achieved at a reasonable cost.

Providing a reliable communication mechanism between independent clock domains is a well known problem. Traditionally, inputs that are asynchronous to a particular clock domain are synchronised by passing the input signal through two flip-flops, as described previously. The probability of failure due to metastability not resolving in time may be reduced to an arbitrarily small number by increasing the time provided for its resolution [125]. Here a trade-off is made between reliability and latency.

An alternative and completely reliable approach is to stretch the clock of the module receiving data when additional time is required to resolve metastability [20, 164, 93]. In this case the clock must be generated locally using a pausible ring oscillator [94, 164]. Limits to the performance of such an approach are imposed by the requirement that the clock period must be larger than twice that of the clock tree insertion delay. This is necessary to ensure that no more than a single clock edge is present in the clock tree at any one time. This guarantees that the clock may be paused when necessary [164, 127]. Unfortunately, high-performance designs already require that multiple clock edges are generated before the first edge appears at the outputs of the clock tree. This problem is compounded by the need to gate clock signals in order to reduce power dissipation (clock gating), which also leads to an increase in insertion delay.

For the highest performance systems, such as a superscalar processor, GALS techniques may be difficult to apply. Traditional synchroniser designs represent a minimum synchronisation delay of at least two clock periods, an increase in this delay may also be necessary as clock frequencies rise. Generating pausible very high-frequency clocks on-chip with the necessary characteristics would also be difficult. One possible approach would be to reduce clock tree delays by increasing the number of clock domains, although the latency penalty involved in communicating between a large number of clock domains may be too large. A more realistic approach may be to simply generate clock domains with clock frequencies which are either equal or at integer multiples of each other. The remaining uncertainity in the phase of each clock can then be tackled by adaptive synchronisation techniques [34, 53]. This approach would also require interconnect to be routed from each clock domain to a central phase detector, which limits the scalability and performance of such an approach.

2.5 The impact of process scaling

Figure 2.5 shows the recorded reduction in feature size for CMOS processes over the past fifteen years and the SIA [1] predictions for the next fifteen years. Historically each successive generation has offered improvements in the speed, density, and power consumption. As technologies continue to scale the ability to exploit potential gains with current design methods is becoming increasing difficult. Much of the renewed interest in asynchronous circuits is a result of these challenges and the potential advantages asynchrony offers. This section provides an overview of process scaling and its impact on the choice of control paradigm.

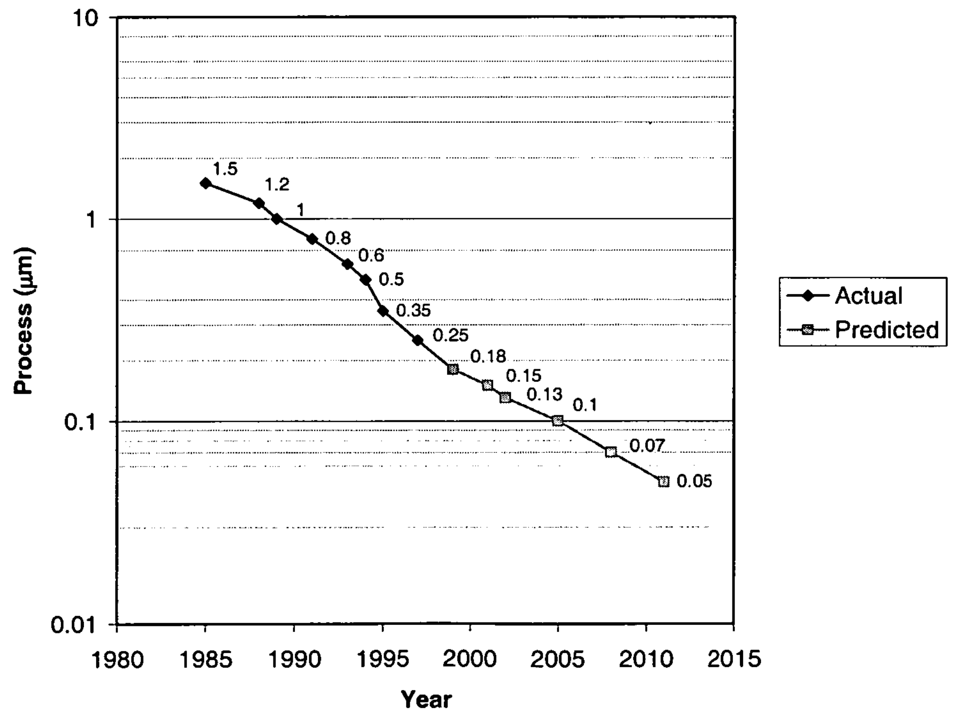


Figure 2.5: Process Generations

Each new generation of CMOS technology provides around a 30% reduction in the lateral and vertical dimensions of a transistor, reducing capacitances and hence switching time. Performance improvements due to technology alone have accounted for around a 30 – 35% increase in microprocessor performance per year [62, 19] since the mid 1980s.

Dynamic power consumption is also reduced as the capacitance switched falls.

$$Power = C \times V^2 \times f \tag{2.1}$$

As a consequence a 30% reduction in capacitance translates to an equivalent saving in power dissipation. In practice faster switching times are exploited to enable designs to operate at a higher clock frequency. In ported designs where the supply voltage remains constant and the clock period is reduced by 30%, power requirements remain constant. An alternative to *constant voltage scaling* is to lower the supply voltage in addition to feature sizes (*constant field scaling*). A 30% reduction in supply voltage results in a 50% reduction in power, even after the clock period is scaled. The range of supply voltages used in each process technology is shown in Figure 2.6. Traditionally lower voltages have been used in power critical applications such as battery powered devices. When performance must be maximised higher voltages have been advantageous.

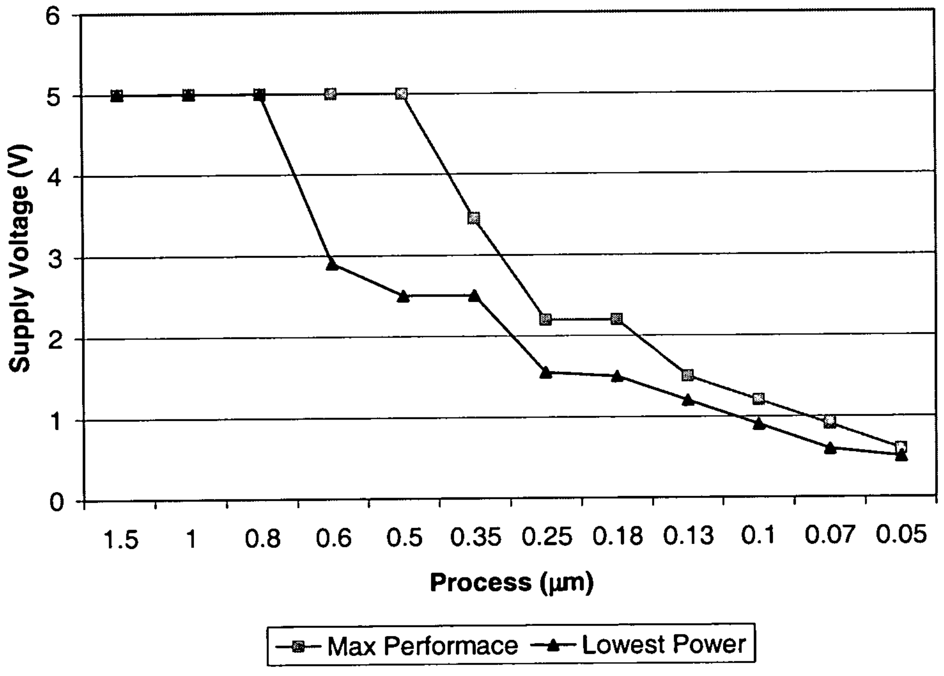


Figure 2.6: Voltage Scaling. Values after $0.18\mu m$ are predictions [1]. Recorded results are taken from major microprocessor generations.

Figure 2.7 plots the actual and predicted power requirements of high performance microprocessors. The trend is in fact one where power requirements are increasing even after supply voltage is scaled. This is a result of new processor designs which exploit greater numbers of transistors and which accelerate the increases in clock frequency. One example of this is the use of deeper pipelines by reducing the number of logic levels required per pipeline stage. Greater levels of ILP, made possible through speculation and dynamic scheduling, also lead to increased power requirements as a result of higher resource utilisation. These rising requirements power will force a steady convergence of supply voltages to their minimum, as illustrated in the graph.

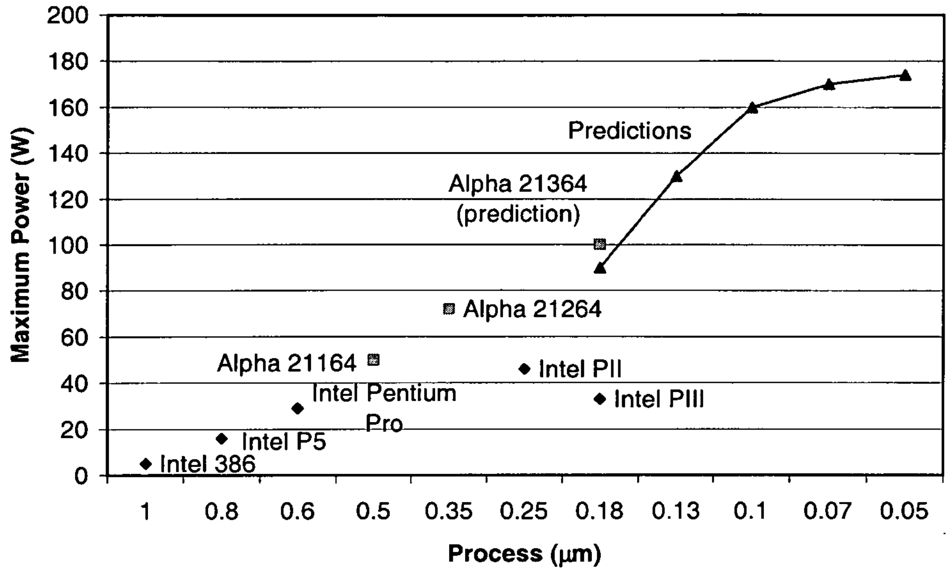


Figure 2.7: Power Dissipated by a selection of microprocessors and predicted values for future high-performance designs [1].

Large power requirements pose problems both due to the heat that must be removed and the supply of power to the die. Dissipating more than around 30W of power begins to add sharply to the total integration cost [145]. Hot spots in a design also limit performance; this problem will be compounded by the negative impact of high temperatures on static power dissipation [19]. It has already been seen that reducing supply voltage alone is inadequate in preventing increases in power, in response recent microprocessors have begun to include other approaches to lowering power.

The largest component of CPU power is currently consumed when transistors switch; as a result many low power techniques attempt to reduce unnecessary circuit activity. The fact that circuits are active even when they are not taking part in useful work is often due to the clock. Isolating inactive subsystems from the clock by *clock gating* [115] is one technique used to circumvent this problem. *Signal gating* is also used on non-clock signals for similar reasons, *e.g.* the inputs to a multiplier may be isolated when it is not in use. While clock and signal gating are used successfully to reduce power consumption in synchronous circuits, their use complicates both circuit design and timing analysis [58]. In particular additional logic in the clock tree complicates the analysis of skew. Explicit control is also required for each module whose clock is gated. For these reasons the granularity to which these techniques may be applied is limited [145]. In contrast, the event-driven nature of asynchronous circuits allow them to naturally avoid superfluous circuit activity. Clock gating may also introduce large cycle-to-cycle power fluctuations placing additional requirements on the power supply. The concentration of switching activity at the point the system is clocked also produces similar problems, requiring careful design to minimise any resulting voltage drops [56]. Asynchronous circuits, in the absence of global synchronisation, may spread switching activity more evenly over time, thus alleviating such problems. This also aids in the reduction of electro-magnetic radiation emissions [47, 109, 29].

Static power requirements due to leakage currents are also increasing. Subthreshold leakage currents increase exponentially as a result of reductions in threshold voltage, which are required to maintain switching times as the supply voltage is lowered. Gate oxide leakage is also increased as gate oxide thickness is reduced. Figure 2.8 (reproduced from [145]) shows a first order analysis of these leakage trends.

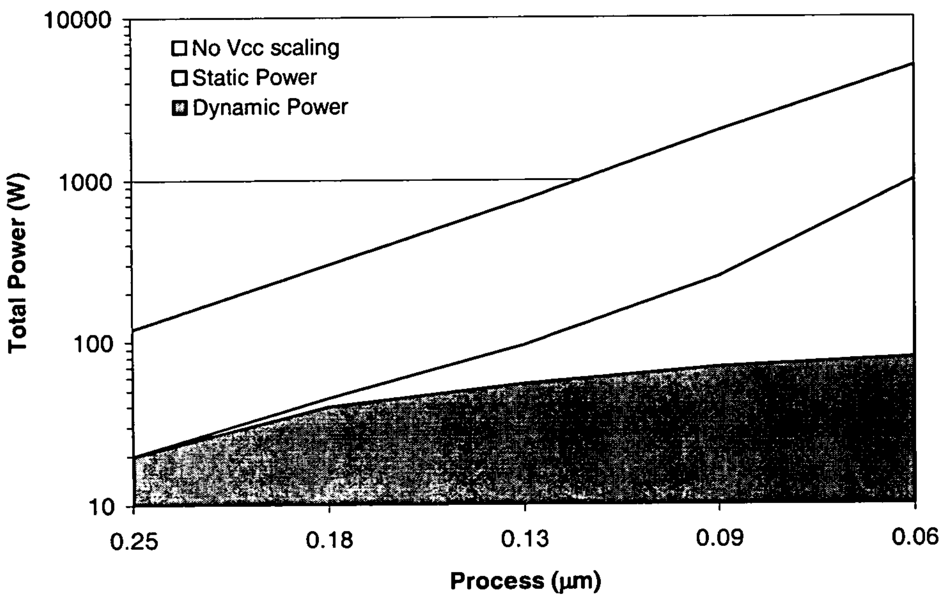


Figure 2.8: Static and Dynamic Power Trends. Trends for static and dynamic power are calculated using constant field scaling. The total power requirements for the case where voltage remains constant is also shown.

The increased importance of both dynamic and static power consumption also makes the need to minimise worst-case delays increasingly costly in a synchronous design. As critical paths determine clock frequency, they must be optimised however rarely they are followed. In previous process generations where gate-delay has been the overriding performance limiter, the addition of extra gates to reduce critical path delays has not been too costly. A potential advantage of asynchronous circuits is their ability to exploit average-case delay as power limits both the number of transistors and transitions. Implementations are possible which sacrifice worst-case delay to achieve a far simpler implementation with equal or improved average-case performance. Examples include, adders and dividers [87, 51, 105, 156], and other complete datapaths [18, 166, 15, 103]. The fine-grain control over resource allocation may also help in reducing resource requirements. While resources must be budgeted for a worst-case scenario in the synchronous case, asynchronous systems can potentially operate with fewer resources, if adequate buffering in the system can provide smoothing for peaks of high utilisation.

The continuing increase in the performance of synchronous systems also depends on the generation of high-frequency clocks. This in itself is a complex design problem requiring significant design resources [34, 40]. Minimising clock skew is a major problem in the presence of supply voltage and process variations. Recent designs have required adaptive de-skewing circuits to minimise skew due to on-die variations [34]. Distributing sharp clock edges and minimising skew also requires large clock buffers, with substantial power requirements. The generation of the global clock signal (even before conditional and local clocks are generated) often account for the single highest use of power. For example the global clock network in the Alpha 21264 microprocessor accounts for 32% of total power requirements. As clock periods decrease, multiple clock periods must exist in the clock distribution network simultaneously, further complicating design. Alternative approaches to clock distribution have also been explored. Wireless clock distribution using integrated CMOS receivers has been demonstrated at frequencies approaching 10GHz [42]. Similar approaches using optical injection are also being explored [98].

Interconnect, together with transistor dimensions, is also shrinking. By reducing the length of wires their overall delay falls in line with gate delay. In the case where wire lengths cannot be reduced, delays will not scale as the overall resistance of the wire increases as the area of the cross-section is reduced. Replacing aluminium interconnect with materials with a lower resistivity, such as copper, is one way to lower the delay of global interconnects. Replacing the traditional silicon dioxide interlayer dielectric with materials with a lower dielectric constant will also bring improvements. Another possibility is to increase the width and height of wires to reduce their resistance, the so called *negative scaling*. The requirement for fat wires leads to hierarchical wiring solutions and increased levels of wiring. The most dense level of wiring is available at the bottom while the uppermost layers contain fatter wires suitable for global interconnect [142]. Of course, interconnect performance is ultimately limited by the speed of light, regardless of the actual implementation of the communication mechanism.

Reducing metal interconnect spacings while maintaining taller wires to reduce resistance also exacerbates the problem of cross capacitance between neighbouring wires. The result of such *crosstalk* is to alter the interconnect delay. If the transitions on neighbouring wires are in the same direction, the delay will be reduced, while transitions that occur in opposite directions increase delay. The impact on delay is determined by how close the transitions occur and their slew rates [72]. Operating asynchronously has the advantage that transitions are not forced to occur together, hence providing an opportunity to reduce the impact of crosstalk. The use of delay-insensitive signalling schemes also allows variations in interconnect delay to be exposed and exploited. The encoding scheme itself may also reduce crosstalk by reducing the number of neighbouring data wires which change simultaneously [12]. In a synchronous environment, the insertion of inverters to stagger signals and balance the effect of aggressor signals can help reduce crosstalk, at the cost of increasing time of flight. Other schemes focus on the provision of shielding. In both the synchronous and asynchronous cases crosstalk noise has the potential to change logic values and lead to failure.

Large variations in interconnect delay, increasing power usage and the difficulties in maintaining global synchronisation (low skew) at high-frequencies have led to suggestions that multiple on-chip clock domains will be required. The use of asynchronous logic to implement a global interconnect between such frequency domains has also been predicted [1]. From a power perspective, the possibility of selecting from a range of clock frequencies aids in making judicious use of the available power budget. Where performance goals can be met at a lower clock frequency, either due to higher levels of available parallelism or lower requirements, a lower clock frequency will save on dynamic power. Dual V_t processes can again make similar savings in static power dissipation if lower performance can be tolerated. Increased levels of integration and design reuse also suggests that designs from different sources will be integrated, requiring support for a range of clock frequencies. The need for multiple frequency domains also requires the generation of multiple clock signals and their distribution. A latency penalty must also be paid when communicating between different clock domains due to the synchronisation overhead.

2.5.1 Summary

The previous section described many potential advantages of adopting an asynchronous approach. It is also true that techniques are being developed which enable synchronous designs to mimic many of these advantages, but their application is usually at the cost of increased complexity and design time. The most significant reason for a shift to an asynchronous approach would be in order to reduce this complexity. The biggest gains will come in providing clear interface specifications free from timing assumptions and a significant easing of the reliance on accurate physical timing information.

2.6 Asynchronous Processor Architecture

This section describes the problems in attempting to adapt synchronous pipelined architectures to operate asynchronously. Much of the previous work in this area is also reviewed. The introduction of asynchronous control influences processor design at the architectural level. The greatest impact comes from the lack of implicit knowledge about datapath state. Control decisions can only be made with state information available locally, or with data obtained through explicit communications.

2.6.1 A synchronous instruction pipeline

Pipelining divides an instruction's execution into a number of distinct stages. The execution of a number of instructions is then overlapped by allowing instructions at different stages of execution to progress concurrently. Figure 2.9 illustrates a possible organisation of a five stage pipeline.

In such a pipeline, the register file represents the lowest level of the memory hierarchy and provides the basic mechanism for naming operands and communicating results. In the absence of pipelining, all the values stored in the register file are up to date when each instruction begins to execute. In a pipelined architecture, data values are read and written at different stages; as a consequence, it may no longer be the case that results are immediately available from physical registers.

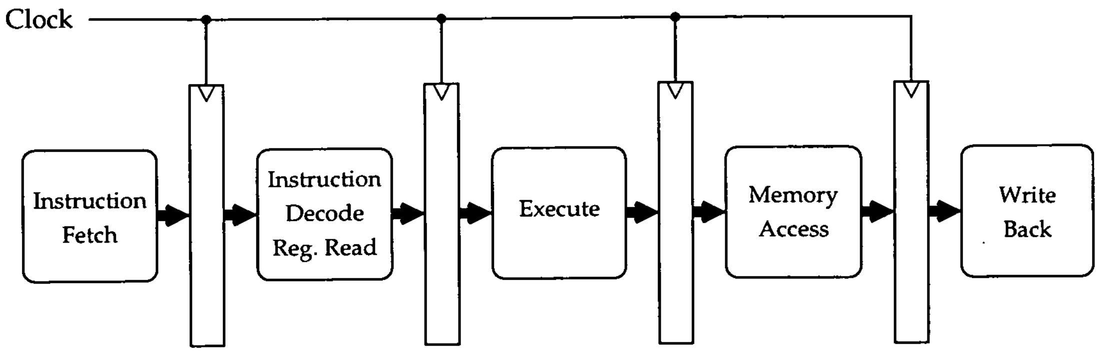


Figure 2.9: A typical RISC instruction pipeline

Consider the example in Figure 2.10. At time t+2 the second instruction will attempt to read the result generated by the first. In this case, the data obtained from the register (R1) will be incorrect as the first instruction will not write its result to the register file until time t+4. The correct value will not be available from the register file until time t+5.

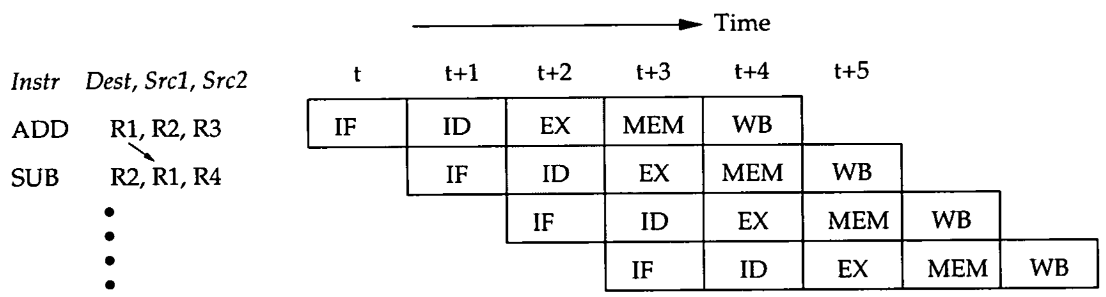


Figure 2.10: A pipeline hazard (true dependency). IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Write Back.

If no other communication mechanism could be provided within the pipeline, such dependencies would force instructions to be stalled or rescheduled at compile-time. To overcome this problem, additional buses are usually provided to bypass the register file, allowing results to be sent directly between pipeline stages. This mechanism is called *data-forwarding*. Figure 2.11 illustrates the various buses used to forward results to the input of the execute stage for a single operand.

The multiplexor, at the input to the execute stage, selects one of three possible sources of data: the register value read in the previous stage, the result of the last instruction (ALU), or the result of the instruction before that (ALU or memory). Results which are read from the register file on the same cycle as they are written are obtained through *bypasses* in the register file. An alternative to providing bypasses is to order writes and reads within a single clock cycle. Memory instructions require two execute cycles (address calculation and memory

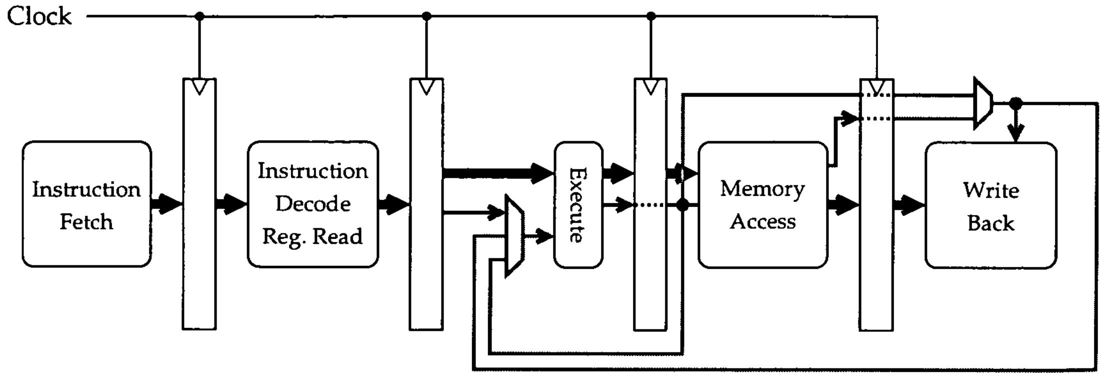


Figure 2.11: An instruction pipeline with result forwarding

access) which means that the result of a load is not immediately available to the next instruction even with data-forwarding. This situation is usually handled by stalling the pipeline, known as a pipeline or hardware interlock, or by introducing *load-delay slots* into the program [62].

By operating synchronously a snapshot of the datapath state at the start of the current clock cycle is always available to the control logic. This makes it trivial to determine the correct source of data for a particular operand. The lockstep operation of both datapath and control logic also guarantee that the data at the selected source will always be valid. For example, if an instruction I enters the execute stage we can assume that instruction I-3 must have completed write back and that the data read during instruction decode is valid.

2.6.2 An asynchronous instruction pipeline

The previous section has shown how synchronous architectures are able to exploit implicit state information in the implementation of both register-based communication and data-forwarding. The elastic nature of an asynchronous pipeline prevents any assumptions being made by one pipeline stage about the state of another. The implicit state information available through global synchronisation must be reproduced by explicit communication or maintained as local state information.

Register based communication may be extended for use in an asynchronous environment with the addition of a locking mechanism. The *register locking* mechanism allows the reading of a particular register to be stalled until pending writes have completed. A simple locking scheme is implemented by appending an extra bit to each physical register. Each instruction's destination register can now be locked by setting this bit during decode, which is reset and the register unlocked after the instruction's write-back operation has completed. Any reads that are made when the register is locked are stalled. The issue units of early supercomputers, including the CRAY-I, used the same scheme to enforce dependencies (here the bits were called *busy bits*) [153].

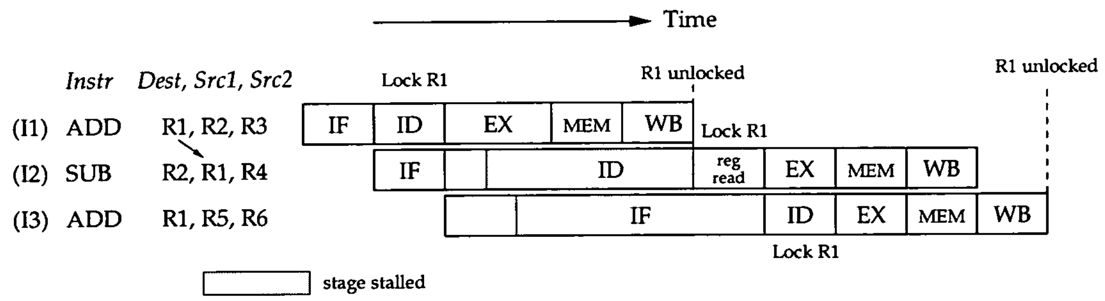


Figure 2.12: Register locking in an asynchronous pipeline. IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Write Back.

The mechanism is illustrated in Figure 2.12. Instruction I1 locks its destination register during instruction decode. Instruction I2 is subsequently stalled when it attempts to read register R1 until I1 has completed its write back operation.

The lock bit is associated with the execution and completion of a single instruction. In a vanilla RISC pipeline this is usually adequate. In the example, instruction I3 is prevented from locking its destination register too early by the intermediate instruction I2. If this was not the case, the potential exists for I3 to lock register R1 when it is still locked by I1. This would result in instruction I1 clearing the lock before I3 had completed. Where such intermediate instructions cannot be guaranteed, *e.g.* when conditional execution is supported, additional lock bits may be required. An alternative is to stall each instruction until the lock bit corresponding to their destination register is clear. This also ensures correct operation in the presence of out-of-order write-backs [117].

The lock FIFO has been proposed as a scheme for organising multiple lock bits [110]. Destination register identifiers are decoded and stored in the lock FIFO. The lock status of a particular register is determined by obtaining the logical OR of all bits in the column of the FIFO associated with the register. Figure 2.13 shows the logic associated with each register to obtain write and read select signals. The scheme allows a register to be locked a number of times and also generates write select signals during write back. An interesting feature of the queue is how entries propagate in the FIFO (micropipeline) without invalidating the lock bit, this is achieved by temporarily holding entries in both the succeeding and preceding stage.

Allowing multiple instructions with the same destination register to be in-flight simultaneously may in general complicate the process of issuing instructions and writing back results, while also limiting ILP. Register renaming (see Section 2.8.2) removes these problems by providing each instruction with a unique physical destination register.

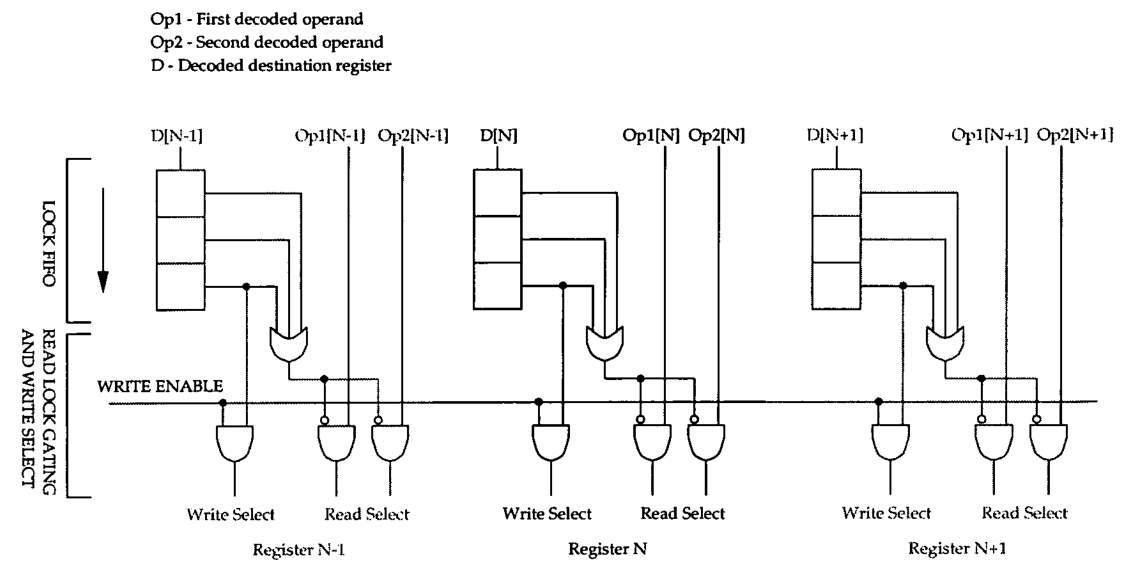


Figure 2.13: The lock FIFO

To maintain performance a data-forwarding scheme must also be incorporated into the asynchronous instruction pipeline. The challenge presented by an asynchronous pipeline is that we can now longer predict where a particular instruction or result is in the pipeline. The communications necessary to identify the location of a result would in fact synchronise the operation of pipeline, mitigating many of the advantages of the asynchronous approach.

A number of different asynchronous architectures are reviewed in the following sections. Each uses a different approach to handling dependencies in an asynchronous environment. Many organise their functional units in parallel in an attempt to increase their utilisation, which may be useful even in scalar machines if functional unit delays vary and out-of-order write-back is possible.

2.6.3 Amulet Processors

The Amulet group at the University of Manchester have designed and fabricated a number of asynchronous microprocessors since 1990 [45, 47, 46]. Three ARM compatible microprocessors have been developed with increasing levels of performance and functionality. The latest AMULET3 design is said to have broadly the same performance as a synchronous ARM9 processor operating at 120Mhz (manufactured in the same technology).

From an architectural perspective, each design has included a more sophisticated approach to handling dependencies. The AMULET1 relied solely on the lock FIFO as described in the Section 2.6.2 to ensure dependencies are respected. The development of AMULET2 aimed to improve performance. In addition to a move from 2- to 4-phase handshaking, a form of data-forwarding was also introduced. Designs have generally used a bundled-data implementation style.

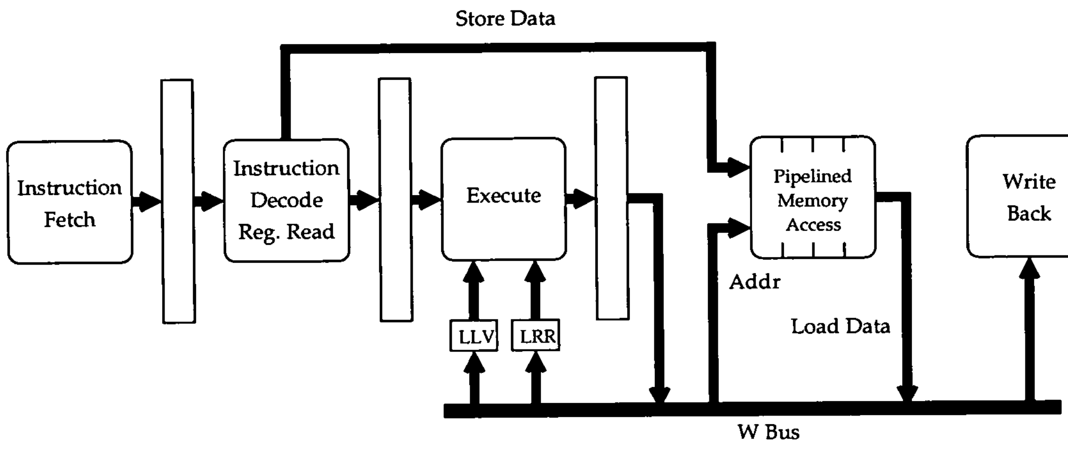


Figure 2.14: AMULET2 style pipeline

A simplified block diagram of the AMULET2 organisation is shown in Figure 2.14. In the generic RISC pipeline described in earlier sections, all instructions were forced to spend at least one clock cycle in every pipeline stage. An alternative to this is to skip pipeline stages when they are not required. In the case of AMULET2 the memory access pipeline may be bypassed. This behaviour is advantageous as it prevents the availability of independent ALU results depending on the completion of earlier memory operations. A consequence of this type of organisation is the need for out-of-order write backs. For this reason a separate lock FIFO is used for internally generated results and loads from memory. The possibility of write-after-write (WAW) hazards may be avoided by stalling the issue of an instruction until its destination register is unlocked (in the other FIFO).

The forwarding of results in the AMULET2 was achieved using a technique called *last result reuse*. This allows the previous result of either an ALU or memory load operation to be forwarded to the input of the execute stage.

The use of the previous ALU result is relatively straightforward as we know that it is always available. This is due to the fact that only when the previous ALU operation has completed can a new instruction enter the execute stage. In the

AMULET2 scheme an additional register, called the *Last Result Register* (LRR) is used to hold the previous ALU result. The reading of the register file can be bypassed by detecting the availability of data from the LRR during decode. This simply requires the decode stage to store the previous instruction's destination register. The use of the last loaded value from memory is more complicated as we cannot assume that its contents is always up to date or valid.

The *Last Loaded Value* (LLV) register is updated each time data returns from memory. Opportunities to make use of the forwarding register and bypass the register file are again identified in the decode stage. In this case, the destination register of the last load instruction is stored and compared to the operands of subsequent ALU instructions. An additional constraint in the case of the LLV is that each time an ALU instruction is decoded with the same destination register the LLV's contents can no longer be used. The AMULET2 architecture also introduces the possibility that there may be multiple outstanding memory operations. To ensure that the latest memory value is present in the LLV register a small FIFO is used to record the progress of pending memory operations. Only when the FIFO is empty can the contents of the LLV be read.

In practice, in the AMULET2e test chip [48] it was discovered that the other parts of the system including the decode stage, address interface and memory, limited the overall performance of the design. This made the resulting performance impact of the inclusion of such forwarding schemes small.

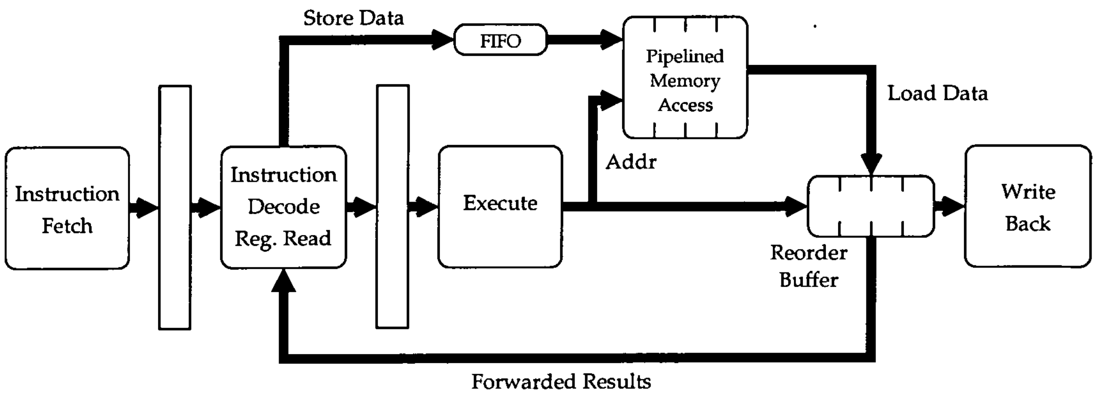
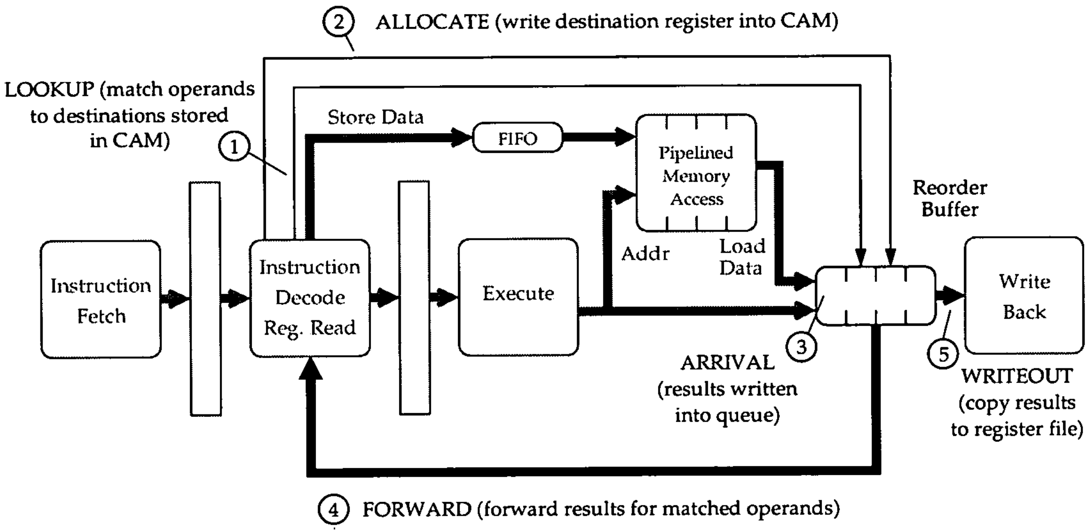


Figure 2.15: AMULET3 style pipeline

The AMULET3 design includes a more complete data-forwarding capability. The use of a reorder buffer [129] (see also Section 2.8.1) provides the possibility of forwarding data from the previous N results, regardless of WAW hazards. The reorder buffer also provides a mechanism for ensuring that all write-backs to the register file are in program order. This sort of mechanism is used extensively in synchronous designs which exploit out-of-order issue and completion. Maintaining in-order program state in the register file is important here to support precise interrupts. In the absence of such a mechanism, as is the case in the AMULET2 architecture, operations that follow a memory access must be prevented from completing until exception detection has completed, potentially imposing a large performance penalty [52]. An outline of the AMULET3 architecture is given in Figure 2.15.

The reorder buffer implementation is based around a *parallel FIFO* [161, 160]. A small RAM in the buffer is indexed by two pointers to create a circular buffer. Once a write has been completed the location of the new buffer entry is static, this is in contrast to a micropipeline style FIFO where each data item always propagates through every FIFO stage. In addition to reducing latency, this characteristic is important in providing random access to data within the buffer.



2 ALLOCATE (write destination register into CAM)
LOOKUP (match operands to destinations stored in CAM) 1
Store Data
FIFO
Pipelined Memory Access
Reorder Buffer
Instruction Fetch
Instruction Decode Reg. Read
Execute
Addr
Load Data
Write Back
3 ARRIVAL (results written into queue)
5 WRITEOUT (copy results to register file)
4 FORWARD (forward results for matched operands)

Figure 2.16: Reorder Buffer Processes

Figure 2.16 illustrates the five processes which access the reorder buffer. Potential operand sources are identified by the *lookup* process[1] during instruction decode. This matches the current instruction's operands to the destination registers of those instructions already allocated entries in the buffer. These comparisons are performed in parallel using a Content Addressable Memory (CAM). The bit mask produced by this process indicates possible data sources and is used by the *forwarding* process. The second step, *allocation*, is also performed during decode. This reserves an entry in the buffer for the result of the current instruction and writes the destination register address into the corresponding CAM entry. The allocated buffer address is carried by the instruction and used during the *arrival*

[1]In this dissertation the word *process* is used to refer to the component or group of components responsible for carrying out a particular task.

process to write the result to the correct buffer entry. The *writeout* process simply waits for results to become available in the original program order and copies them to the register file.

The reorder buffer operates as a cache for recently generated results. A result may be forwarded from the buffer as soon as it arrives and is available until every other buffer entry has been allocated. The allocation process must then reuse the entry, forcing the data to be obtained from the register file. A detailed description of the operation of the reorder buffer within the AMULET3 is given in [52].

2.6.4 Micronet Processors

The idea of a micronet as an operational model for distributing control in an asynchronous architecture was developed at the University of Edinburgh [9, 8]. Micronets model a processor, even a scalar one, as a network of functional units which compute concurrently and communicate asynchronously. In [117] the study of the conversion of a synchronous processor architecture to an efficient micronet-based one was undertaken. The process was described in a number of refinement steps, each introducing additional scope for exploiting fine-grain parallelism and decentralising control.

In order to maximise the utilisation of datapath resources, the architecture is designed to enforce only minimal constraints on their use. This is best achieved by breaking each individual instruction into a number of micro-operations. Each micro-operation is now scheduled independently, which allows different micro-operations from different instructions to exist in the same logical pipeline stage. For example, the architecture exposes resources such as register read ports and operand buses to allow different instructions to use them concurrently. Minimum constraints on the scheduling of such micro-operations are explored together with their implementation. The idea of exposing micro-operations to allow them to execute concurrently under local control is an important one. This idea forms the basis for the architectures described in Chapters 3 and 4.

The development of scheduling algorithms targeted at micronet processors was also explored [10]. The work also suggested additional schemes for exploiting information obtained at compile-time; for example, the use of concurrency bits to aid instruction issue. The theme of exploiting a close interaction between compiler and architecture is continued in the work described in Chapter 4.

An instance of a RISC architecture developed using the micronet model is shown in Figure 2.17. Arbitration to access the single write back bus is in this case handled by a small token ring. WAW hazards are avoided by ensuring that

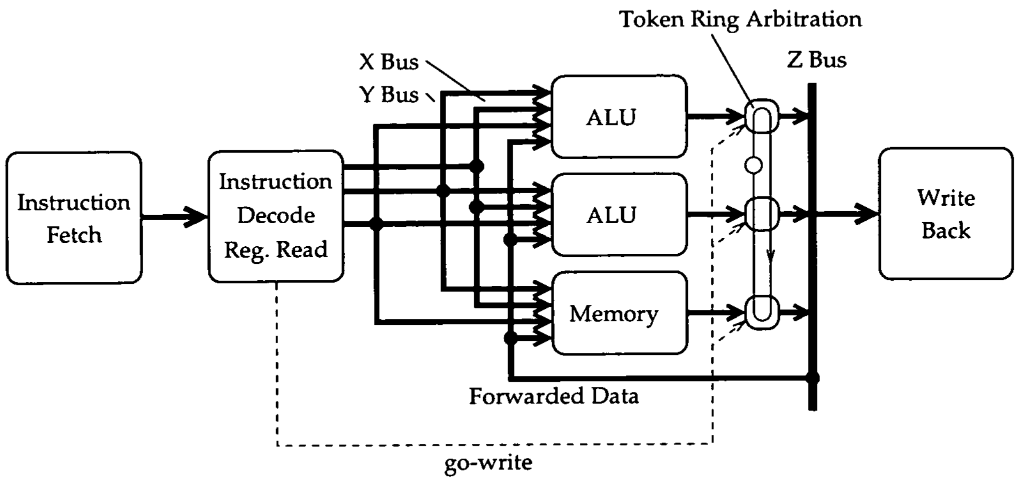


Figure 2.17: A Micronet architecture

an instruction's destination register is unlocked before write back can take place. In the case of a WAW hazard the architecture avoids stalling instruction issue and instead only stalls the write back operation itself. The *go-write* signal shown in the diagram is generated at the register file and indicates when it is safe to write back a particular result.

Forwarding is implemented by tagging each result with its destination register identifier. When a write-back is taking place the tag may be matched to operands currently being fetched. This allows the operand fetch stages to obtain the data from the write-back bus before the result is available from the register file. In such an event, a second handshake signal sent to the register file cancels the pending read request.

2.6.5 MiniMIPS processor

The MiniMIPS [88] processor was designed and fabricated at Caltech between 1995 and 1998. An outline of the architecture is shown in Figure 2.18. Although a number of parallel functional units are supported, the design lacks a renaming mechanism such as the reorder buffer described previously. Results are simply reordered by polling functional units in the order they were used. This technique is similar to the use of a *result shift register* [130], and is the simplest way in which precise interrupts may be supported.

The results of functional units that complete out-of-order are unavailable until all previous instructions have written their results to the register file. As a consequence, forwarding of data is only possible from one instruction to its immediate successor (in program order). The cases when forwarding can take place are detected during decode by maintaining a record of the previous instruction's destination register. As both the writing of results and reading of operands is performed in program order, there is no possibility of Write-After-Write (WAW) or Write-After-Read (WAR) hazards.

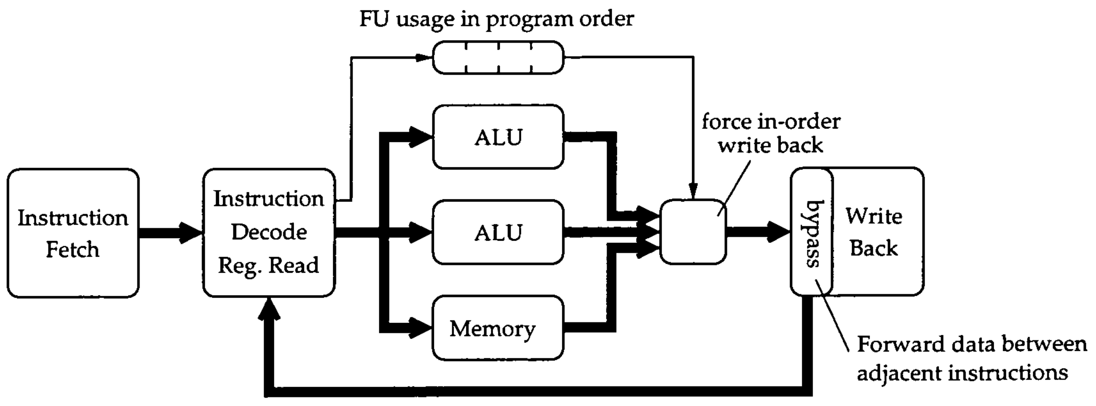

FU usage in program order
ALU
ALU
Memory
Instruction Fetch
Instruction Decode Reg. Read
force in-order write back
bypass
Write Back
Forward data between adjacent instructions

Figure 2.18: Minimips style pipeline

At a lower level, the design benefits from a number of innovations, which include the use of pipelined completion detection, pipelined caches [106], and the design of a low-latency adder. Unlike the AMULET processors described earlier, the design is implemented using full-custom dynamic logic. Techniques to optimise the number of pipeline stages and buffering, while guaranteeing correctness are also presented [82, 85].

The performance of the processor is reportedly high when measured in MIPS (approx. 165MIPS@3.3V in a $0.6\mu m$ technology). Unfortunately it is unclear what impact data and control dependencies have on performance. Therefore peak MIPS must be considered a very bad indicator of performance in the case of a deeply-pipelined asynchronous processor. A more useful measure of performance could be obtained by running benchmark programs and reporting their execution times.

2.6.6 Hades Architecture

A simplified view of the Hades architecture [38] is presented in Figure 2.19. In addition to writing results to the register file, each functional unit maintains its last result in a special forwarding register. Opportunities to forward results from these registers are subsequently detected during instruction decode. This simply requires a comparison between the current instruction's operands and the destination addresses of the results currently allocated to the forwarding registers. This operation is similar to the lookup operations, performed during instruction decode. This operation may again be performed using a small Content-Address

Memory (CAM). Each time a new instruction is issued, this CAM is updated to reflect the new data which will become available. An *overwrite* signal is also sent from the decode unit to the corresponding functional unit to indicate that a new result may be written into the forwarding register. This ensures that results are only overwritten after any forwarding operations have been completed.

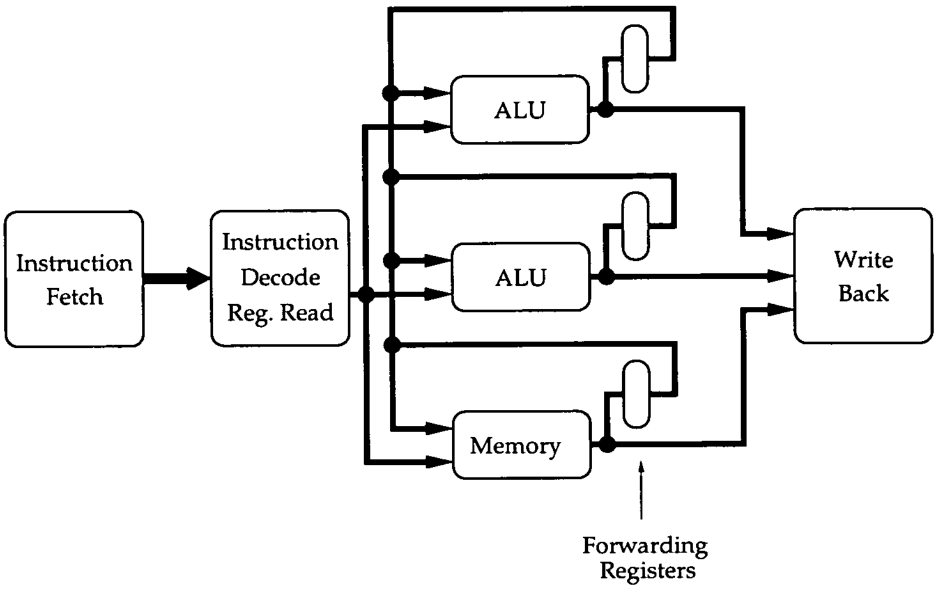


Figure 2.19: Hades style pipeline

A simple example of how this forwarding scheme operates is provided below. Events are described from the point at which the first instruction listed in Figure 2.20 obtains its operands.

```
(1) LD R1, [R2]
(2) ADD R3, R1, R4
```

Figure 2.20: Example Program Fragment

1. The decode unit updates its CAM to indicate that the memory unit will produce a result destined for register $R1$.

2. An overwrite signal is generated to permit the contents of the memory unit's forwarding register to be overwritten. This also indicates that the current contents are out of date.

3. The first instruction is dispatched to the memory unit and begins execution.

4. The second instruction enters the decode stage. A match in the CAM indicates that register $R1$ should be available from the memory unit's forwarding register.

5. The second instruction is issued and a forwarding request is made to the memory unit. If the load operation has not yet been completed, forwarding will be stalled until the result is available from the forwarding register.

6. The result of the load is eventually received by the ADD instruction together with its other operand. This allows the instruction to progress to its functional unit.

Register writes and reads are synchronised using a register locking mechanism, employing a single lock bit per register. The lack of a renaming scheme forces WAW hazards to be avoided by stalling instruction issue. Only when an instruction's destination register is unlocked may it be allowed to assert the lock and issue. This restriction and the possibility of only forwarding from the last instruction issued to each functional unit, means that data forwarding is not supported in all cases (as is the case in the AMULET3's reorder buffer). In practice, the scheme may be generalised to allow the forwarding of the previous N results generated at each functional unit. One way in which this may be achieved is described in the next chapter.

2.6.7 Alternative Pipeline Organisations

The processors described in the previous sections are all loosely based on the RISC pipeline described in Section 2.6.1. In such architectures two mechanisms exist for obtaining operands: the register file and data forwarding. The fact that there are multiple potential sources of data is more problematic in an asynchronous implementation than a synchronous one. An alternative is to implement a single communication mechanism for all results, irrespective of their age.

Figure 2.21 illustrates a number of possible ways of organising instruction and data flow. Option (a) represents the traditional RISC pipeline. The figure shows a number of parallel instruction pipelines, instructions obtain operands by reading the register file and write results back to the register file after execution. Conceptually instructions and results flow in the same direction, except for the special case when the possibility to forward data is detected.

An alternative organisation (b) is to provide results from a single source, in this case a counterflowing result pipeline. This allows all data to be obtained

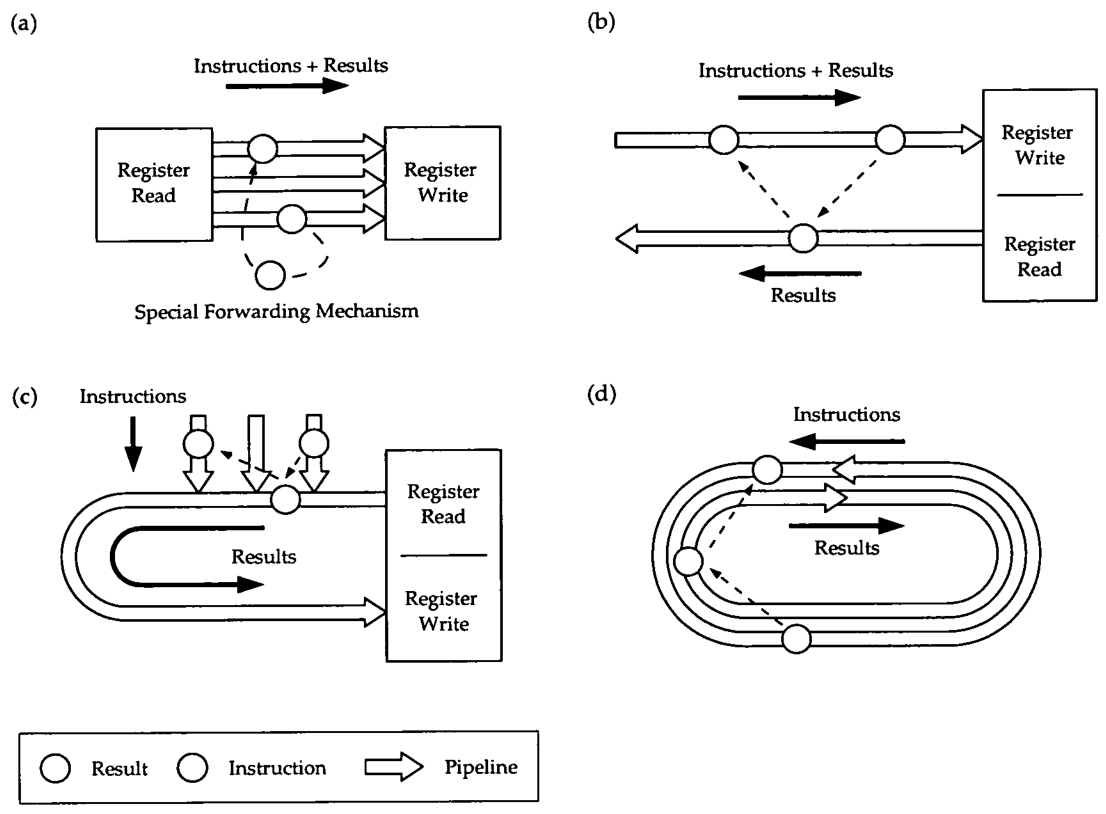


Figure 2.21: Instruction and Data Flow Possibilities

by a local communication between the instruction and result pipelines. Results are also carried in the instruction pipeline and are eventually committed to the register file. Register read operations may be initiated during instruction decode to ensure that all the required operands are present in the result pipeline. The third possibility (c) also maintains a single operand source. In contrast to (b) instructions are placed in separate execution pipelines. Results are also returned to the register file using the same result pipeline that is used to provide operands. A purely data-flow organisation is shown in (d), here instructions flow around their pipeline until all their operands and a functional unit of the required type are ready.

A number of architectures which exploit these alternative pipeline organisations have been presented since the development of the Counterflow Pipeline Processor by Sproull, Sutherland and Molnar. The counterflow pipeline is an example of organisation (b). Examples of (c) and (d) include the Rotary Pipeline Processor and Counterdataflow processor. Each of these architectures are described in the following sections.

2.6.8 Counterflow pipeline architecture

Localising control and communication is advantageous in both the design of asynchronous systems and the use of DSM technologies. This idea together with the aim to develop a simple modular processor architecture based around the concept of micropipelines, led to the development of the Counterflow Pipeline Processor (CFPP) [136].

Within the processor a result pipeline provides a means of obtaining all operands through local communication. To make this possible, instruction and result pipelines flow in opposite directions. Each time an instruction produces a result it is able to insert it into the result pipeline for later instructions to acquire. Results are also carried to the end of the instruction pipeline where they are written into a register file. Operands are read and sent down the result pipeline after requests are made during instruction decode. Figure 2.22 illustrates one possible counterflow organisation.

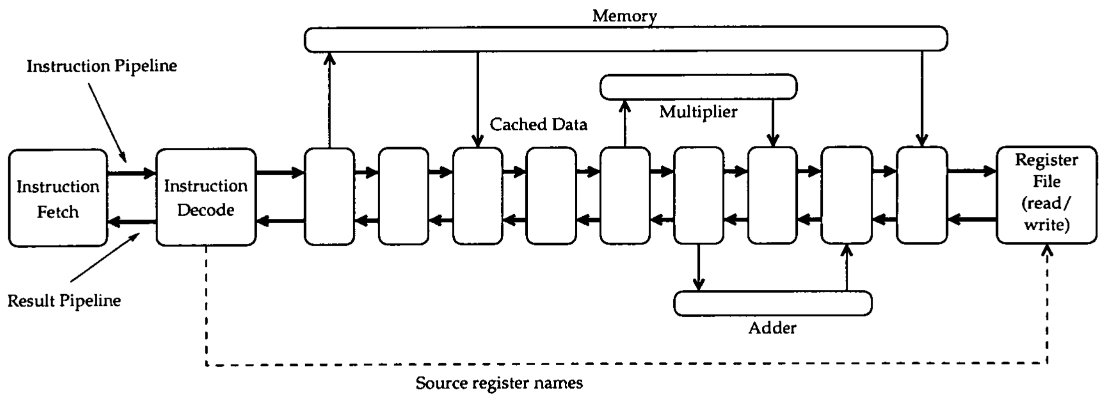


Figure 2.22: An example of a counterflow style pipeline

Functional units reside in sidings which may be accessed at predetermined stages in the instruction pipeline. At one stage an instruction, once it has received its operands, will dispatch an operation to the functional unit. A number of stages later the result and instruction will merge and continue until they reach the register file. In this way in-order write-back is maintained. Careful design of the local control at each stage is required to ensure that results cannot move past an instruction without the opportunity for the result to be required, a process called *garnering*.

2.6.9 The non-stalling Circular counterflow architecture

Work described in [89, 67], describes the development of a non-stalling counterflow or counterdataflow architecture. Their architecture attempts to solve three problems that they identify in the original CFPP concept. Firstly, the potentially high latency required to fetch from the register file. This is a result of the register file and issue stages being at opposite ends of the pipeline. Secondly, if instructions stall in the original CFPP all successive instructions may also be forced to wait. Stalls may be a result of the need to wait for access to the last FU of a particular type. Finally, extending the architecture to issue multiple instructions in parallel was considered difficult due to the complexity of detecting cases where instructions could safely be issued together.

The first of these problems was tackled by moving the register file to the bottom of the pipeline, next to instruction decode and issue. The need to stall instructions was then prevented by feeding the end of the instruction pipeline back through the decode unit to create a ring. This always allows instructions to progress with the knowledge that they will eventually reach a free FU of the correct type. The process of dispatching an instruction to a functional unit now removes the instruction permanently from the instruction pipeline. The multiple issue of instructions is also simplified as all data dependencies are resolved in the pipeline.

The performance limitations of all the counterflow-based architectures described are discussed later in Section 2.6.14.

2.6.10 Rotary pipeline processor

The rotary pipeline [92] exploits a number of general purpose data buses organised as a ring. The buses provide a means of transporting results in all communication scenarios. In addition to providing a direct path for communication between functional units, they also provide a means of reusing data values fetched for one instruction, if required as operands by another. The basic organisation of the rotary pipeline is shown in Figure 2.23. The connection of a particular bus to an input or output of a functional unit is controlled by a set of switches as illustrated.

An instantiation of the architecture without an explicit register file is also possible if adequate register storage can be made available in the result ring alone. The architecture differs from the counterflow architecture described in the previous section by allowing instructions to be issued directly to their functional units. This simplifies the local communications required between the result buses and functional units.

The effectiveness of such an architecture hinges on the ability to allocate buses in an efficient manner. Managing a table indicating the allocation of data to particular bus segments centrally seems unrealistic. Buses would probably have to

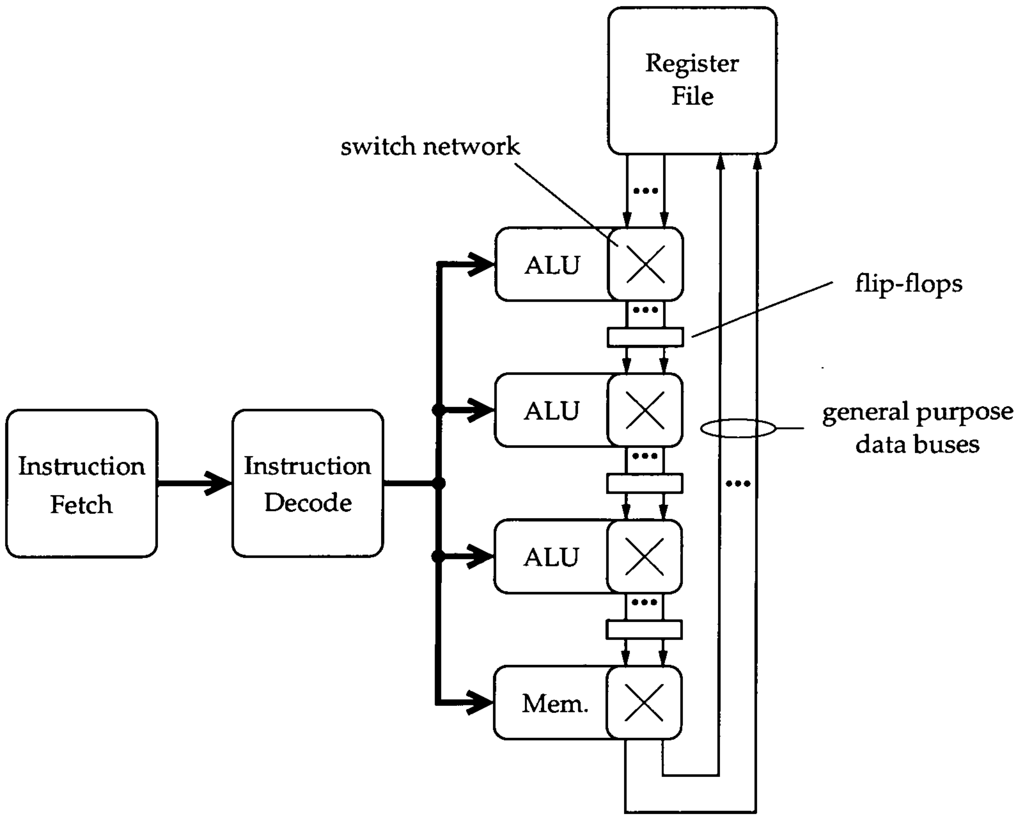


Figure 2.23: The Rotary Pipeline

be allocated in round-robin fashion to avoid the need for synchronisation between the dispatch and write-back processes. Control of individual bus segments could be handled locally, using a series of segment interfaces organised as a micropipeline. Opportunities to forward data (results or register operands) from one bus segment to the input of a functional unit would have to be determined during dispatch, in a similar fashion to AMULET3 or Hades. The allocation of buses for write-back is complicated by the fact that it cannot be determined ahead of time whether a particular operand may be reused by a subsequent instruction. Although where possible, it is probably a sound policy to always reuse an operand bus as a write-back bus. It must also be noted that the architecture does not provide uniform support for forwarding. Only the function units downstream are allowed to receive the new data directly.

The use of dedicated buses in a traditional superscalar or RISC architecture simplifies their allocation and ensures that latencies are minimised.

2.6.11 Alternative Instruction Set Architectures

The architectures in the previous sections name operands and specify the destination of results by directly addressing registers in a register file. The existence of a large number of *general-purpose registers* is important in such architectures for two reasons. Firstly, pipeline architectures often benefit from both local and global instruction scheduling. Given enough registers, the way in which dependencies are represented using a register file provides few constraints on scheduling. Secondly, the register file is the lowest level of the memory hierarchy, providing a fast multi-ported memory to store both intermediate results and distribute those results required by different instructions. Some systems also extend the register file to provide an indexed addressing mode for registers. An example is the *rotating register-files* [116], which provide a restricted form of register renaming which may be used to support software pipelining.

Operands may also be obtained implicitly; common examples include *accumulator* and *stack* architectures. Some form of implicit naming may also be exploited in addition to a register file; for example, the use of *register-windows* or result *queues.*

The register file provides a means of communication through a shared memory. Alternatively, communication may be specified explicitly within the instruction (analogous to a message-passing scheme). In this case each instruction must specify where its result will be used in the datapath. This scheme is adopted by the *SCALP* processor described in the next section. Other examples of this type of approach include Transport Triggered Architectures (TTA's) [30], where data-transports also trigger functional-unit operations. An asynchronous version of a transport triggered architecture was verified using CCS in [49].

2.6.12 SCALP: A Superscalar Low-Power Processor

The development of a Superscalar Low-Power Processor (SCALP) is described in [39]. A register-less implementation was devised based on *explicit forwarding.* The hope was to reduce power consumption by increasing code density and decreasing the overall complexity of the processor. Each instruction specifies explicitly the location where its result is required. An example of how this may work in practice is given in Figure 2.24.

Unfortunately, non-deterministic behaviour introduced by control hazards and asynchronous operation complicates the use of such a mechanism. For example, it may be necessary to introduce explicit sequencing instructions in some cases to guarantee the order particular results arrive at a functional unit. Communi-

```
x = y * 30 + c        load [y_addr] -> mul_a
                      Mul 30 -> alu_a
                      load [c_addr] -> alu_b
                      Add -> ....
```

Figure 2.24: Example of code generated for SCALP

cation across branches is also problematic as the destination of the result cannot be determined *a priori*. In this case a register bank functional unit is used. Duplicate instructions are also introduced into the program to allow results to be distributed to more than one destination (the duplicate and sequence instructions are executed in the *move unit*). The need to send many results through the register file and move functional units actually results in lower dynamic code density for the benchmarks tested.

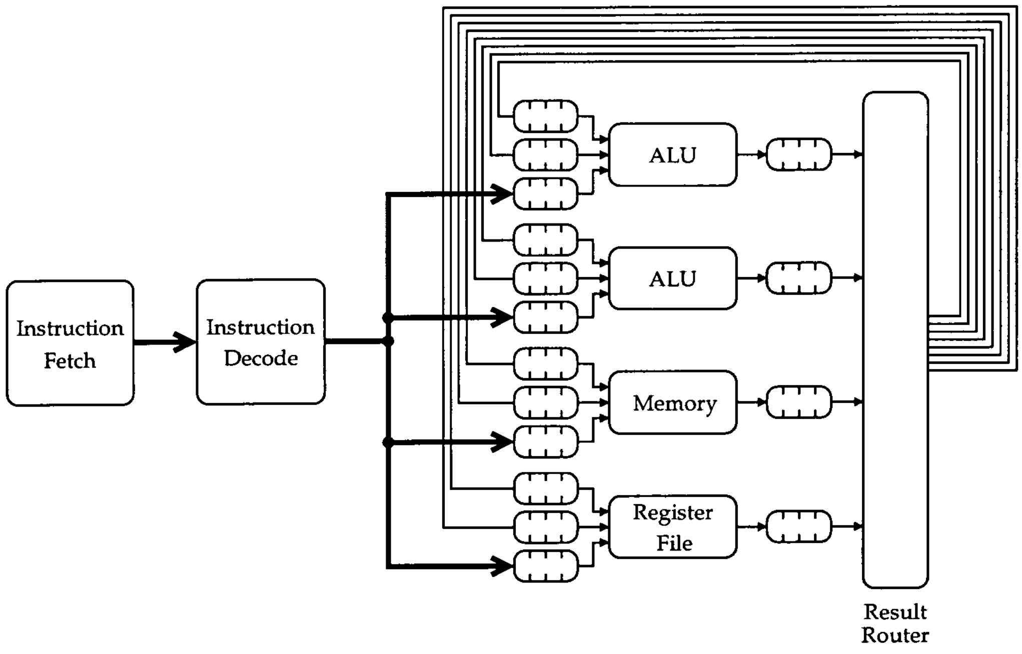


Figure 2.25: The SCALP architecture

The SCALP architecture is shown in Figure 2.25. Queues provide buffering for both instructions and results, reducing the need to stall the issue unit and the operation of functional units. Overall performance is said to be lower than expected, due to a combination of poor code density and the inability of the architecture to expose and exploit instruction-level parallelism.

2.6.13 Other Asynchronous Processors

Implementations of the TITAC architecture, based on the MIPS R2000, have been fabricated at the Tokyo Institute of Technology [102, 101]. Although asynchronous circuits are used throughout the design, the processor operates in a completely pseudo-synchronous manner at the architectural level. As a consequence, a register locking mechanism is not required and a synchronous style forwarding mechanism may be adopted. The ECSTAC [95, 4] processor developed at the University of Adelaide is also designed to operate completely asynchronously. Register locking is implemented using a single lock-bit attached to each register. The architecture does not support data-forwarding.

Various asynchronous implementations of micro-controller architectures have also been designed and fabricated. These include Philips' 8051 microcontroller [50] and an asynchronous version of Cambridge Consultants' XAP processor at the University of Cambridge. Due to the modest degree of pipelining in such architectures, data-forwarding is not usually beneficial.

2.6.14 Performance limitations of existing approaches

Many of the architectures described in the previous sections have made attempts to exploit asynchrony. In most cases this has required some modification of the instruction pipeline, either to reduce the impact of synchronising communications or to better exploit datapath resources. The handling of data-dependencies in particular has produced a number of novel solutions. The alternatives presented may be broadly categorised as follows:

Local Forwarding All results are made available at some point in time through a local communication. No explicit management of forwarding is performed. Examples include the rotary and counterflow pipelines.

Centralised Forwarding The possibility of forwarding is detected during instruction decode. Results are subsequently obtained from a forwarding register or buffer. Examples of this organisation include AMULET2/3 and Hades. A further distinction may be made between architectures that centralise the temporary storage required to hold values that may be forwarded and those that distribute the registers amongst the functional units. Such a forwarding mechanism exploits the fact that the order in which instructions will be executed is known during decode.

Explicit Forwarding The possibility of forwarding data is indicated explicitly by the program. This removes completely the synchronisations typically required to determine if forwarding is possible. The technique is used in the SCALP architecture and in the instruction compounding technique described in Chapter 5.

While many of the designs present solutions to the problem of handling data-dependencies in scalar processors, the techniques do not necessarily provide efficient solutions for architectures attempting to exploit higher levels of ILP.

Local forwarding schemes tend to suffer from the need to sequentialise both the flow of instructions and results, as a result it has been acknowledged that they are unlikely to be competitive with existing high-performance architectures [29]. Implementations often also require excessively wide datapaths. While attempts have been made to overcome these problems, such architectures still tend to extend operand fetch latency, which severely limits the levels of ILP that may be exploited. Developing dynamic scheduling schemes that recirculate instructions is undesirable both due to power requirements and resulting irregular dispatch latencies. One potential pitfall of developing architectures while targeting an asynchronous implementation is to favour an elegant implementation while sacrificing overall performance.

Both the centralised and explicit forwarding schemes offer the potential to provide efficient asynchronous implementations without requiring a completely novel processor architecture. Many desirable architectural features such as, parallel operand fetch and parallel functional units may be retained. Of particular interest are schemes that do not require results to be stored temporarily in a central buffer. This avoids the need to limit forwarding performance by the need to write and read results to and from a slow central structure such as a reorder buffer. In a superscalar architecture this is likely to have a much greater access time due to the requirement for both a large number of read/write ports and entries.

The use of both centralised and explicit forwarding schemes in asynchronous superscalar architectures will be explored further in the following Chapters. We also explore how dynamic scheduling complicates the implementation of the data forwarding scheme further and how attempting to solve each problem in isolation leads to a poor overall solution.

2.7 ILP Architectures

A typical RISC pipeline exploits the parallelism available between different phases of an instruction's execution. In order to make full use of the available ILP requires an architecture that allows greater numbers of instructions to execute concurrently. The availability of independent operations may then be exploited in one of two ways. Firstly, deeper pipelines may be introduced to make better use of resources over time (*temporal parallelism*). Secondly, operations may be performed in parallel by duplicating resources (*spatial parallelism*). Architectures that exploit ILP using deep pipelines are often called *superpipelined*, while those which are able to fetch and execute multiple instructions in parallel are called *superscalar* or *Very Long Instruction Word* (VLIW). It may be noted that increasing the degree of pipelining or parallel issue should in theory provide similar performance gains [70]. In practice, careful trade-offs must be made between the degree of pipelining and the number of instructions fetched per cycle. Pipelining is ultimately limited by latching overheads and the ability to evenly distribute delay between different stages. The majority of modern high-performance architectures can in fact be considered both superscalar and superpipelined, maintaining high-clock frequencies with deep pipelines and exploiting many parallel datapaths.

Superscalar architectures differ from VLIW architectures in their ability to schedule operations at run-time. While this allows them to exploit information only available at run-time, it also requires additional hardware to identify operations that may be performed in parallel. In contrast, a strict VLIW micro-architecture is simplified by the fact that the compiler guarantees all instructions that are fetched together are independent. The schedule will also guarantee that the required functional unit is free. If a suitable instruction cannot be found at compile-time a *no-op* may be inserted. Although this results in conceptually simpler datapaths, expected gains in terms of power or performance may in practice be difficult to realise. Power may suffer due to the need to perform greater speculation in order to maintain competitive performance, while performance itself is limited by the quality of very complex optimising compilers. Dynamic scheduling hardware in contrast may compete with compile-time global schedules even when they are able to predict branches perfectly [81]. In addition, practically all of the current high-performance processors have demonstrated that superscalar implementations are possible with both high clock rates and dynamic scheduling hardware.

Figure 2.26 (from [116]) illustrates a number of different possibilities for partitioning operations between the compiler and micro-architecture. These range from a purely superscalar approach, relying mainly on hardware scheduling of both instructions and resources, to a VLIW approach where both instructions and the majority of hardware resources are allocated during compilation.

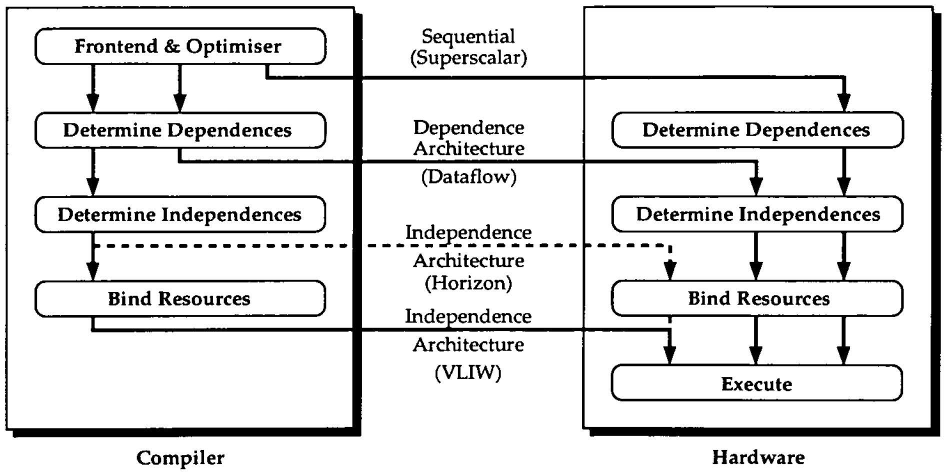


Figure 2.26: Division of responsibilities between the compiler and hardware for three classes of architecture (reproduced from [116])

In general, systems that exploit information provided by the compiler about independent instructions are called *independence architectures*. While a VLIW architecture represents the extreme case where all the instructions fetched are independent, the possibility exists of providing additional bits to indicate parallel sub-groups. Recent examples of this approach include Intel's IA64 architecture [64] and TI's VelociTI VLIW architecture [147], which are both capable of indicating variable size groups of independent instructions. Earlier examples of similar techniques are described in [143, 76, 151]. The SCISM [151] architecture also enables some dependent instructions to be included for parallel issue where a suitable interlock collapsing functional unit exists[2]. A simple example may be where two dependent addition instructions may be issued to a single 3-input ALU. In addition to identifying independent operations, the compiler may also be used to specify forwarding or communication operations explicitly. Examples of architectures which exploit this type of compiler/datapath interface were described in Section 2.6.11. The ability to specify dependencies explicitly also forms the basis for the instruction compounding technique described in Chapter 4.

In the following chapters we discuss the development of a number of asynchronous multiple-issue architectures. We shall in all cases describe them as superscalar as some degree of dynamic scheduling is exploited. The following section

[2]Work presented in [151] also introduces the notion of an instruction compound, this is unrelated to the work described in this dissertation

provides an outline of the architecture of synchronous superscalar processors.

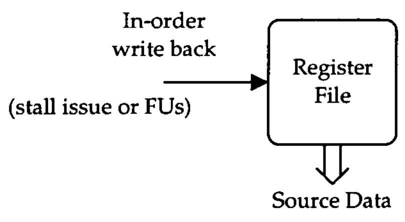
2.8 Superscalar Processors

The design of instruction set architectures is usually based on a purely sequential programming model. The resulting interface between compiler and processor reflects the way instructions are executed in a simple scalar pipeline. If the processor's architecture is extended to become superscalar, this interface essentially remains unchanged, while it becomes possible to fetch and execute multiple instructions in parallel. The utilisation of the duplicated datapath resources is now dependent on exposing the available ILP dynamically. Contemporary designs achieve this by forming a window into the dynamic instruction stream from which instructions may be issued out-of-order. As the ILP available from a single basic block is far too limited, hardware branch prediction is used to extend the scheduling window across multiple outstanding branch instructions. The possibility that instructions may now be fetched and executed from a mispredicted branch direction requires support for the speculative execution of instructions. The efficient use of the instruction window also requires the removal of inter-instruction dependencies, which do not represent real data-dependencies. These so called false- or anti- dependencies are often introduced due to the way operands are named. Support for both speculative execution and the removal of false dependencies is usually provided through a form of register renaming. The register renaming mechanism provides the architecture with the required flexibility in the management of results and provision of operands. Together these techniques are used to create a buffer of instructions that is continuously scanned at run-time in search for instructions that may be dispatched to free functional units. While the aim of the hardware described above is to exploit ILP through speculation and out-of-order execution, additional hardware is also provided to ensure that from a programmer's or interrupt-handling perspective the sequential execution model is retained.

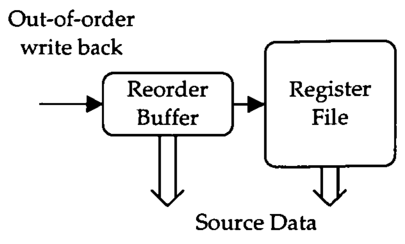
The following sections provide a brief introduction to these mechanisms and an overview of how they are incorporated into a number of different superscalar architectures.

2.8.1 Precise Interrupts

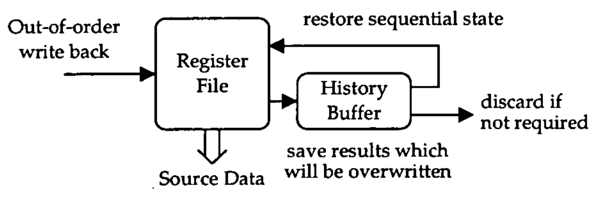
The organisation of a superscalar architecture is heavily influenced by the need to support precise interrupts. The provision of precise interrupts requires that on the event of an interrupt the datapath is restored to a state that is consistent with a sequential model of execution [130]. This enables a minimal amount of state to be saved in order to restart execution after the interrupt has been serviced. It also allows the interrupt to modify the subsequent behaviour of the program; for example, in the case of a page fault or arithmetic exception. For architectures that exploit out-of-order write-back or more complex techniques such as dynamic scheduling, the provision of this state information requires additional state-sequentialising hardware.



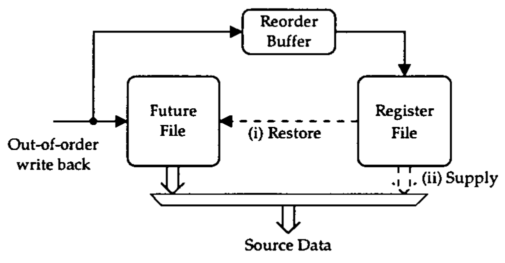
(a) In-order write back



(b) Reorder buffer



(c) History buffer



(d) Future file

Figure 2.27: State-serialising hardware

Figure 2.27, illustrates four possible techniques for providing a sequential state from the register file. The first technique (a) is to simply force all register writes to occur in program order. This may be achieved by allocating cycles on the write-back bus prior to issuing an instruction. Instruction issue is stalled when it is not possible to ensure that register writes occur in the correct order [130]. A similar scheme is employed in the asynchronous MiniMIPS architecture (see Section 2.6.5). In this implementation the order in which register writes are performed is controlled locally at the register file preventing the need to initially stall instruction issue. In general, forcing in-order write-back reduces ILP by serialising the availability of results to subsequent instructions. For this reason the technique is never used in superscalar designs.

The reorder buffer (b) provides a technique that not only ensures that the register file is updated in program order, but also provides access to results generated out-of-order. The mechanism operates by allocating an instruction and its future result an entry in the buffer prior to issue. As results are generated they are written out-of-order to their corresponding entries in the reorder buffer. Entries are subsequently held in the buffer until all prior results have been committed in program order. The buffer provides a temporary store for results generated out-of-order, allowing execution to continue unhindered. An asynchronous reorder buffer providing support for both precise interrupts and forwarding is described in Section 2.6.3.

One potential drawback of the reorder buffer scheme is that operands may now reside in one of two locations: in the register file or temporarily in the reorder buffer. The process of determining the location of a particular result in the buffer involves an associative lookup prioritised by the age of the instruction that produced it. The buffer must also be implemented as a multi-ported memory in order to permit parallel read and write accesses. This additional complexity may be undesirable if a large number of instructions are issued in parallel, or if a large number of instructions may be in-flight simultaneously. Even though these complications exist, a reorder buffer is usually adopted in some form in a superscalar design. In practice, the problems of identifying results in the reorder buffer and the supply of operands may be simplified through the use of the register renaming mechanisms described in the next section. For completeness, two earlier schemes that attempted to overcome these problems and which probably influenced later schemes supported by explicit register renaming are described below.

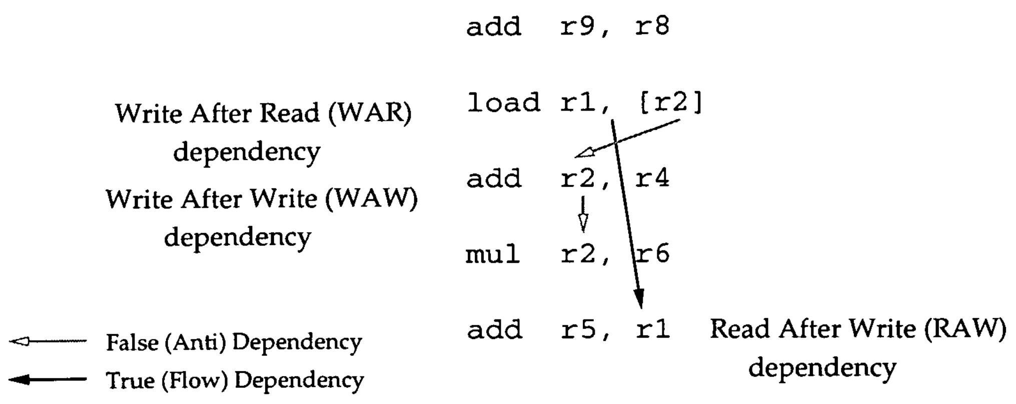
The use of a history buffer [129] (c) provides a mechanism for restoring the sequential state of the register file in the event of an interrupt. During normal program execution results are written out-of-order to the register file, which can as a result provide all source data. The history file is similar in structure to a reorder buffer maintaining instructions in their original program order. As each instruction writes to the register file, the data it displaces is written into its corresponding entry in the history buffer. The buffer contents may subsequently be used whenever an interrupt occurs to restore the register file to the required sequential state. Unfortunately, while the scheme provides a single source of operands it requires multiple clock cycles to restore the state of the register file. The final scheme (d) overcomes this problem by maintaining two complete register files. The first, called the *future file* is updated with out-of-order write-backs and provides a source of operands during normal operation. The second, provides the sequential state maintained by a reorder buffer. The dotted lines distinguish two possible organisations, (i) the case where the future file is updated to by the register file after an interrupt [130] or (ii) the possibility of selectively reading the operands from either register file in order to minimise the delay associated with restarting after an interrupt [68]. In this case, each entry in the future file also contains a valid tag indicating if the contents should be read or the register file output selected. In the event of an interrupt all entries in the future file are marked as invalid.

2.8.2 Register Renaming

The mechanisms in the previous section exploit the provision of additional physical registers in order to provide support for precise interrupts. In general, the number of physical registers in a superscalar architecture is often greater than the number of logical registers available to the compiler. This is important to ensure that false dependencies are removed and that enough state information is maintained to recover quickly after a mispredicted branch. The term register renaming [74] is used to describe the techniques that manage these additional physical registers.

Renaming operates by selecting a new destination for the result of each instruction. All subsequent references to the result of the instruction are then directed to this new physical destination register. The aim is usually to provide a unique destination for each of the instructions that may be in-flight simultaneously. This results in the removal of false-dependencies, which are introduced simply by reusing a logical register name at compile-time. Figure 2.28 provides an example of each possible type of dependency. These false dependencies require that otherwise independent instructions are executed sequentially in order to maintain correct program semantics. Increasing the number of logical registers available to the compiler doesn't necessarily solve this problem as false-dependencies may be introduced between different iterations of a loop. False dependencies, if not removed, limit ILP and complicate the process of determining when an instruction is ready to issue.

The reorder buffer described in the previous section has already introduced one form of renaming. Here additional registers are provided to store those results that are generated out-of-order. The correct source of data for each operand is obtained prior to issuing an instruction by searching the contents of the reorder buffer. The search itself is usually executed in parallel using a content-addressable memory (CAM), if the search results in multiple matches the result corresponding to the youngest instruction must be selected. Entries in the buffer are released and reused once the instruction is able to commit its result to the register file in



add r9, r8
load r1, [r2]
add r2, r4
mul r2, r6
add r5, r1
Write After Read (WAR) dependency
Write After Write (WAW) dependency
Read After Write (RAW) dependency
False (Anti) Dependency
True (Flow) Dependency

Figure 2.28: Data Dependencies

program order.

Register renaming may also be implemented by maintaining explicit register mapping tables. These tables record the current logical to physical mapping for each register [97]. The tables may also be duplicated to record a mapping corresponding to different levels of speculative execution, or one that enables a sequential state to be restored in the event of an interrupt. The potential advantage of renaming registers in this way is that the reorder buffer is no longer required to perform an associative lookup to identify operands. Many processor architectures, including the Intel Pentium Pro, II and III, use a mapping table for this reason alone. In these architectures mapping relationships are maintained in the *register alias table*, which is used to locate results in the reorder buffer. An alternative is to merge the rename registers present in the reorder buffer with the existing register file in order to provide a single source of operands. In such a scheme the reorder buffer is retained, but it is no longer required to store or serve results. It is now simply used to maintain an in-order mapping table for use in the event of an interrupt, it also provides the basic mechanism for releasing physical registers back into the free register pool. In such a scheme the last-use of a physical register is only detected when its corresponding logical register is reused as a destination, more sophisticated schemes that are able to allocate physical registers for a shorter period of time are also possible [91].

In general, register renaming schemes are distinguished by the technique used to translate between logical and physical register names and the location of the additional rename registers in the architecture. A detailed description of the range of implementation possibilities is given in [126].

2.8.3 Speculative Execution

Instructions may be executed speculatively by predicting the outcome of a yet unexecuted instruction on which their execution depends. Common examples include branch prediction, value prediction [83] and cache hit/miss prediction [75]. A mechanism to support speculative execution must allow instructions to be re-executed in the event that a misprediction is discovered.

While value prediction techniques are less common, exploiting even modest levels of ILP requires a branch prediction mechanism. Predicting the direction of branches may be performed either during compilation [131, 22] or at run-time using a hardware branch predictor [62]. Each technique attempts to reduce the effect of control dependencies in order to expose greater levels of ILP. We limit the following discussion of recovery mechanisms to those architectures exploiting hardware branch prediction.

In the event of a mispredicted branch, a mechanism is required to allow execution to be rolled back and restarted from the correct branch destination. The performance impact of mispredicting branches must be minimised by ensuring that the recovery mechanism is fast. For this reason the history buffer is unsuitable for a system supporting both precise interrupts and speculative execution due to the multiple cycles required to restore state. Of the remaining schemes the reorder buffer and future file both offer potential solutions. In the simplest scheme, mispredicted branches are only handled when they reach the end of the reorder buffer. Recovery now simply consists of clearing the entire reorder buffer and restarting execution with the correct program counter value. If a future file is used, its contents are marked invalid as in the case of an interrupt. The drawback of waiting until all instructions prior to a mispredicted branch have completed before initiating the recovery process is a loss of performance. Although Johnson [68] reports that this only results in a 4-5% drop in overall performance, modern superscalar designs usually handle mispredictions as soon as they occur. In the reorder buffer this may be achieved by invalidating only the results queued after the mispredicted branch entry. In the case of the future file this operation is complicated by the fact that the results in the reorder buffer cannot be accessed directly to provide operands. If the complete contents of the future file are to be invalidated, the register file must first be updated with the pending register writes in the reorder buffer. Alternatively, entries in the future file that correspond to instructions on the wrongly-predicted branch must be selectively invalidated. For this reason direct implementations of the future file scheme where immediate recovery from mispredicted branches is required are unrealistic.

Schemes to handle speculative execution are also possible using multiple register mapping tables. A particular mapping table is now associated with each branch prediction. The process of restoring state in the event of a misprediction now simply requires the saved mapping table to be restored. The values stored in the active set of physical registers for this table will have remained unchanged as additions to the reorder buffer are only made at the end of the reorder buffer; either when it is determined that the instruction was on a mispredicted path or that the logical destination register has been reused.

2.8.4 Dynamic Scheduling

In most cases, data-dependent control flow and cache behaviour make it impossible to determine an optimal instruction schedule at compile-time. This provides additional scope for exposing ILP at run-time by relaxing the strict program order execution of instructions. This may be achieved in a limited fashion by allowing out-of-order completion, potentially reducing unnecessary stalls as a result of differences in functional unit latencies. Additional performance gains are possible if instructions can be scheduled dynamically. Scheduling at run-time has the advantage that the availability of functional units and results can be monitored in order to dispatch instructions as early as possible.

The dynamic scheduling process itself operates on a buffer of instructions that is maintained between the instruction fetch and execute stages. This buffer may be viewed as a window into the dynamic instruction stream, potentially spanning many basic blocks. The operation of the scheduling hardware may be divided into the following processes, regardless of its actual implementation:

1. **Issue and Write** Instructions enter the window after the instruction fetch stage.

2. **Initialise** At some point, the initial status of the instruction's operands must be determined.

3. **Wake-up** Maintain the status of each instruction's operands as new results are generated.

4. **Selection and dispatch** For each free functional unit select (possibly from many ready entries) an instruction to dispatch.

5. **Removal** After an instruction has been dispatched, remove the entry from the window and reuse it.

From a performance perspective, it is obvious that a primary goal of any synchronous implementation is that it should not increase the overall cycle-time of the machine. It is also important that the scheduling process does not introduce an additional delay between the execution of dependent instructions. If dependent instructions are to issue in consecutive cycles, the *wake-up* operation associated with an instruction should proceed concurrently with its execution. This allows instructions that are dependent on its result to be dispatched at the beginning of the next clock cycle, obtaining the result through data-forwarding. The issues raised by the inclusion of instructions with variable latencies are discussed later in this section.

Initially we will consider an implementation of dynamic scheduling in which neither wake-up nor select processes operate speculatively. With this restriction the need to schedule dependent instructions in consecutive cycles means that the *wake-up* and *selection* (and *dispatch*) processes must be considered as an atomic operation [108]. As a consequence they must be designed to operate within a single clock cycle. When considering the implementation of the other processes, *issue and write* and *initialise*, it is important to ensure that the status of instruction operands is initialised and updated correctly. Two possibilities exist for implementing the *initialise* process. Firstly, a shared memory may be established between the *initialise* and *wakeup* processes. Correct operation is maintained by ensuring that write (wakeup) and read (initialise) operations are performed in that order for each clock cycle. Alternatively, the status of operands may be initialised by examining the current instructions that lie in the window waiting to be dispatched. This requires that the dispatch status of instructions remains unchanged during initialisation.

The following section discusses a number of possible designs and implementation details. In the following discussions, the hardware used to buffer instructions and perform the dynamic scheduling operation will be called the *dispatch buffer*, while individual entries relating to a single instruction will be referred to as *reservation stations.*

2.8.4.1 Implementation Details

The organisation of the reservation stations themselves, requires that a number of initial design decisions be made. These are introduced below.

- **Centralised/Distributed** A single large monolithic window represents the best use of reservation stations, although cycle time restrictions usually forces some division of the window. Designs typically provide separate dispatch buffers for integer and floating point instructions. A fully distributed window would provide localised reservations stations at each functional unit. Other schemes exist where the particular entries in a centralised buffer can only be dispatched to particular functional units. The PA-8000 [79] uses this type of organisation to achieve its cycle time requirements, here odd and even entries of its dispatch buffer are allocated to different sets of functional units.

- **Compacted/Non-Compacted** While instructions are written to the buffer in program order they will be dispatched out-of-order. The most efficient use of the reservation stations is made if entries are reused as soon as they become free. In general, additional logic will be required to compact the buffer in this way. Designs that merge the functionality of both reorder and dispatch buffers will of course want to retain instructions in the buffer even after they have been dispatched.

- **Are results held in reservation stations?** The need for reservation stations to hold result data is dictated by the position of the register read stage in the processor's pipeline. One possible organisation is to read the contents of registers in the stage preceding dynamic scheduling. This forces results required by an instruction and generated after it enters the window to be stored with the instruction in its reservation station.

When new instructions are first added to the dispatch buffer, the initial status of their operands must be determined. The way in which this is implemented is influenced by the location of the rename registers. If all register state (rename and sequential) is stored in a single register file and register renaming is performed using register mapping tables, register status is usually maintained in a *busy-bit table* [162](MIPS R10k). This table holds a single one-bit entry for each physical register. The entry is reset whenever a physical register is placed into the free-register pool. It is subsequently set to indicate that data is available whenever a result is written to the register. Operand status is initialised by reading the corresponding busy-bit table entry once the operands of newly fetched instructions have been mapped. If a different renaming scheme is employed, where no mapping table is maintained, initialisation may require a different implementation. In the scheme described in [79] (PA-8000), initialisation is performed by identifying dependencies that exist between new instructions and those already in the window. This identifies either the rename register file (if a dependency exists) or the register file storing sequential state, as the source for a particular operand. The scheme requires a 5-bit comparator for each new instruction at each entry in the buffer, 3360 in total for a 48-entry buffer. An additional colour bit appended to the register is used to detect the possibility that a result has moved from the rename registers to become part of the sequential state, since the instruction was initialised.

Once an instruction has obtained the current state of its operands and entered the dispatch window it monitors the dispatch of instructions to determine when it is ready to execute. If we imagine a simple processor where all instructions take a single cycle to execute, the wake-up process simply involves broadcasting the destination register of each instruction that has just been dispatched. These destination register identifiers are then compared to each instruction's operand and their status updated whenever a match occurs. A block diagram of this type of wake-up logic is shown in Figure 2.29.

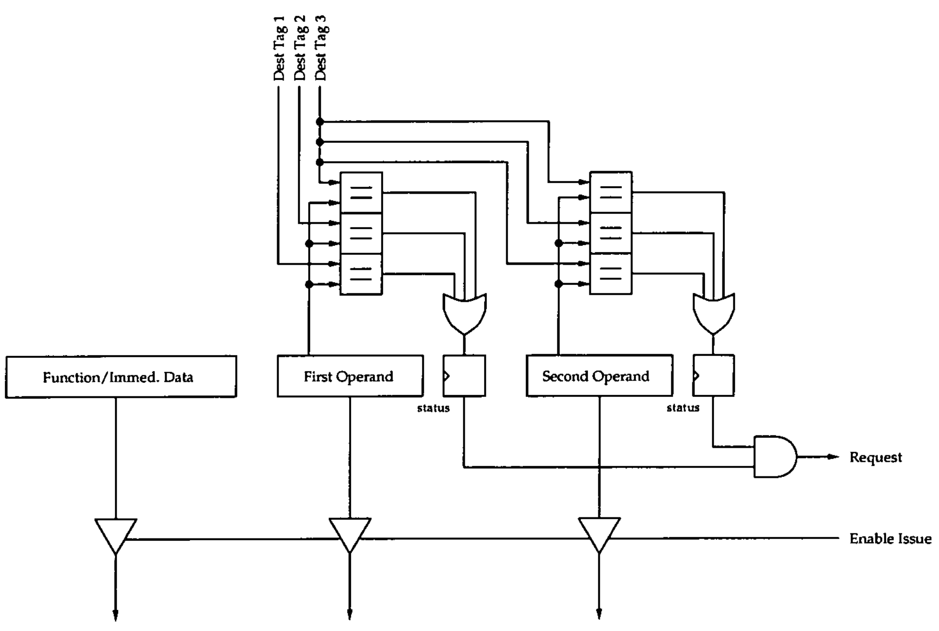


Figure 2.29: Block diagram of wake-up logic for a single reservation station

If it is known that particular instructions require multiple clock cycles in order to produce a result, then the wake-up of the instruction's children should be delayed. In the Alpha 21264, the hardware that tracks the progress of instructions is called the *register scoreboard*. Conceptually, the scoreboard contains a counter associated with each physical register. Whenever an instruction is issued, the counter is set to indicate the latency of the particular operation being performed. On each clock cycle the counter is decremented until the register identifier is broadcast to wake-up dependent instructions in the dispatch buffer.

An additional complication of such a scheme is the need to handle variable latency operations such as loads, which may hit or miss the data cache. In order to maintain performance, instructions dependent on the result of load instructions may be issued tentatively. If, after dispatching the instruction, it is discovered that the load has resulted in a cache miss, then the instruction is squashed and reissued when the data is available. In practice, this may require that many instructions are aborted and a mini-restart of the pipeline is initiated. In order minimise the performance impact of squashing instructions, a load hit/miss prediction table may be used [75, 163]. This can then be used to set the initial value of the wake-up counter to minimise the need to squash and reissue instructions.

2.8.4.2 History and Recent Developments

Dynamic scheduling was originally exploited in the design of early scalar machines that had numerous parallel functional units. Tomasulo's algorithm found in the the IBM 360/91 [146] and the CDC-6600's scoreboarding algorithm [144] are well known examples. The inclusion of dynamic scheduling in superscalar microprocessors became possible in the 1990's with the Metaflow architecture [113]. Other early design examples include the dispatch stack [2, 37], the HPS architecture [66] and the register update unit (RUU) [133, 132]. Johnson's work [68], describes and contrasts a number of these designs.

The latest work in the area of dynamic scheduling focuses on the need to pipeline implementations as the number of logic levels per pipeline stage decreases. Modern microprocessors, such as the Pentium IV, aim to operate at multiple GHz frequencies by employing very deep instruction pipelines (20 stages). If the ability to issue dependent instructions in consecutive cycles is to be retained, the wake-up process now becomes a speculative one. Schemes utilising this type of speculation are described in [137].

2.8.5 Data Memory Accesses

Data dependencies between instructions whose operands are registers can be detected by simply comparing physical register identifiers. For instructions that access memory a comparison of this kind cannot be made until *after* each instruction has made its address calculation. Figure 2.30 provides an example of potential dependencies that may exist between memory instructions in a dispatch buffer. In order to issue the youngest load instruction, it must be ensured that the pending stores in the buffer do not write to the same memory location.

A conservative approach is to dispatch memory instructions twice, once to

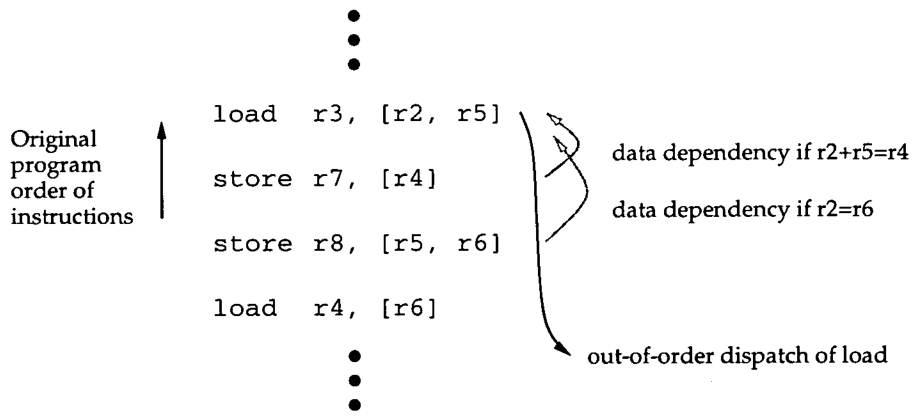

Original program order of instructions
load r3, [r2, r5]
store r7, [r4]
store r8, [r5, r6]
load r4, [r6]
data dependency if r2+r5=r4
data dependency if r2=r6
out-of-order dispatch of load

Figure 2.30: Examples of potential load-after-store hazards in a dispatch buffer

generate an address, and again, after dependencies have been detected to access memory. In such a scheme, memory addresses are stored in the dispatch buffer as soon as they are generated. The dispatch of a load instruction to access memory is now only attempted when the memory addresses of all older pending stores are available. At this point a comparison is possible between the addresses of the pending store instructions and the load. If no dependency is then detected the load is able to access the data memory cache to obtain the correct data. If a dependency is detected and the store data is available, then data may be forwarded between the pending store and load instructions. If no data is available, then the execution of the load must be delayed. Store instructions themselves must be buffered until their speculative status, due to branch prediction or the potential execution of an interrupt, has been resolved.

One possible enhancement is to issue loads speculatively before it can be determined if the load is independent of pending stores. The subsequent detection of a dependency is handled in much the same way as a mispredicted branch, requiring that the load instruction and all subsequent instructions that operated on incorrect data are re-executed. In practice, implementations are usually simplified by forcing an exception when the load graduates, and re-executing the load and *all* subsequent instructions. Loads are handled in this way in many modern microprocessors [75, 79, 162]. In the case of the Alpha 21264 [75], loads that cause exceptions of this type are recorded in a *load wait table.* This allows subsequent exceptions to be avoided by delaying the dispatch of the load until all prior stores have executed. The table is periodically cleared to prevent unnecessary waits. Additional performance gains are possible if load address calculations can be made earlier in the instruction pipeline [11]. More sophisticated techniques for reducing load latency are based on dependency or data-value prediction [148, 84, 96, 122].

2.9 Summary

Asynchronous circuits offer a number of potential advantages as technology scales and the cost of maintaining global synchronisation increases. A major challenge in taking advantage of these benefits is in developing architectures which are able to operate efficiently in an asynchronous environment. Superscalar architectures, in particular, pose a challenge due to the large numbers of high-level synchronisations which are exploited in traditional designs. The following chapters develop and characterise a number of techniques which may be used to produce efficient asynchronous superscalar architectures.

Chapter 3

Towards Asynchronous Superscalar Processors

3.1 Introduction

Architectures that are able to exploit asynchrony should possess two important characteristics: the ability to both expose local variations in delay and translate these local performance gains into an overall improvement in performance. Obtaining these characteristics can usually be aided by distributing control allowing, where possible, for datapath components to be scheduled on a local basis.

Distributing control in this way gives the system the ability to respond to many different potential orderings of events, which provides a framework for exploiting fine-grain parallelism. When supported by a datapath that exposes a large number of independent datapath operations, potential exists for exploiting asynchrony. One major obstacle in organising a superscalar processor in this way is that both dynamic scheduling and data forwarding implementations traditionally exploit global synchronisation. Synchronous implementations rely on the predictable state of the datapath that is guaranteed by its lockstep operation. In an asynchronous system this predictability is often sacrificed in order to improve average-case performance.

A naive asynchronous implementation is one that simply mimics the synchronisations present in a synchronous design. Unfortunately, synchronising operations in an asynchronous system has the effect of exposing the delay of the slowest operation. The impact of requiring regular synchronisations at a high level is that it becomes impossible to exploit performance gains made on a local basis. If performance and timing tends towards the worst-case it is unlikely that an asynchronous design would offer many benefits, as a clock would probably represent a significantly smaller control overhead.

The challenge therefore in devising a good asynchronous implementation is to provide both an effective dynamic scheduling and data-forwarding mechanism, while guaranteeing that individual datapath components have maximum freedom to exploit actual circuit delays.

Provided that such an effective asynchronous design can be devised, the provision of a dynamic scheduling mechanism will potentially complement the scheduling of low-level operations that is already achieved by distributing control. This may be particularly important for an asynchronous processor where the data-dependent nature of delays potentially limits the extent to which a compiler can produce optimal schedules. In addition to delays incurred by functional units, it is also likely that any asynchronous implementation will impose a wide range of inter-instruction communication delays. Related work described in [5, 134] investigates the problems of scheduling for asynchronous targets.

We first introduce a generic superscalar architecture as the framework for exploring a number of implementation possibilities. The areas of the architecture that pose a potential problem for an asynchronous implementation are then identified. This is followed by the development of a dispatch buffer suitable for inclusion in an asynchronous superscalar processor. The addition of a data forwarding capability is then discussed with the introduction of a simplified queue based architecture. The data-forwarding mechanism is then extended to support the fully-asynchronous dispatch buffer. Finally, two other asynchronous architectures that support out-of-order dispatch are discussed: the Asynchronous Fast Dispatch Stack (AFDS)[155] and the FRED architecture [119].

3.2 A Generic Superscalar Processor

In this section we outline the superscalar architecture that will form the basis of the ones explored in this chapter and the next. At this point a number of design decisions can be made that will hopefully simplify the exploration of the design space. These are summarised below.

- Operands are read from the register file after instructions have been dispatched. This simplifies the implementation of the dispatch window in both the synchronous and asynchronous cases, as results do not need to be stored within each instruction's reservation station.

- Operand fetch and the initialisation of operand status can be simplified by maintaining explicit register mapping tables. The existence of a busy-bit table minimises synchronisation by only requiring that a single bit be examined to initialise the status of an operands. The use of register mapping tables also allows all registers to be maintained in a single register file, removing the need to identify the most up to date source for each register.

- The reorder and dispatch buffers will be implemented as separate pieces of hardware, which will provide us maximum flexibility in the design of the dynamic scheduling and execute stages.

- For simplicity, we also assume that the dispatch buffer does not attempt to compact its entries.

This type of organisation is similar to the MIPS R10000 [162]. An abstract view of instruction and data-flow is provided in Figure 3.1, together with an outline of the basic instruction pipeline in Figure 3.2. By examining the paths in Figure 3.1 that feedback to earlier operations we can identify the remote communications that potentially limit the extent to which asynchrony may be exploited.

The communications identified in the diagram do not all represent synchronisations that are problematic in an asynchronous implementation. In many cases the communication will not synchronise the main functions of the pipeline stages. In fact, there are only two places where performance is potentially threatened by the need to synchronise with a remote pipeline stage: firstly, in providing a data-forwarding mechanism and secondly, in initialising and maintaining the state of operands in the dispatch buffer. These problems are examined in more detail in the following sections. A brief description of the remaining remote communications required in the pipeline and why they are less of a problem is given below.

The synchronisation required at the register file is implemented using a register-locking mechanism. Register renaming guarantees that each instruction in flight has a unique destination register. As a consequence a single lock-bit per register is sufficient to implement the locking mechanism. Synchronisation is minimised by performing locking on a per-register basis. Stalls introduced at the operand fetch stage are in practice reduced with the use of a data-forwarding mechanism.

The reorder buffer must also receive destination register identifiers as results are generated in-order to allow instructions to graduate. If the reorder buffer is organised as a parallel FIFO (outlined in following section), the operation of updating the execute status of instructions queued in the FIFO is a simple one. This is a result of the entries within the buffer remaining in fixed locations, allowing the processes that must operate on it to proceed in parallel.

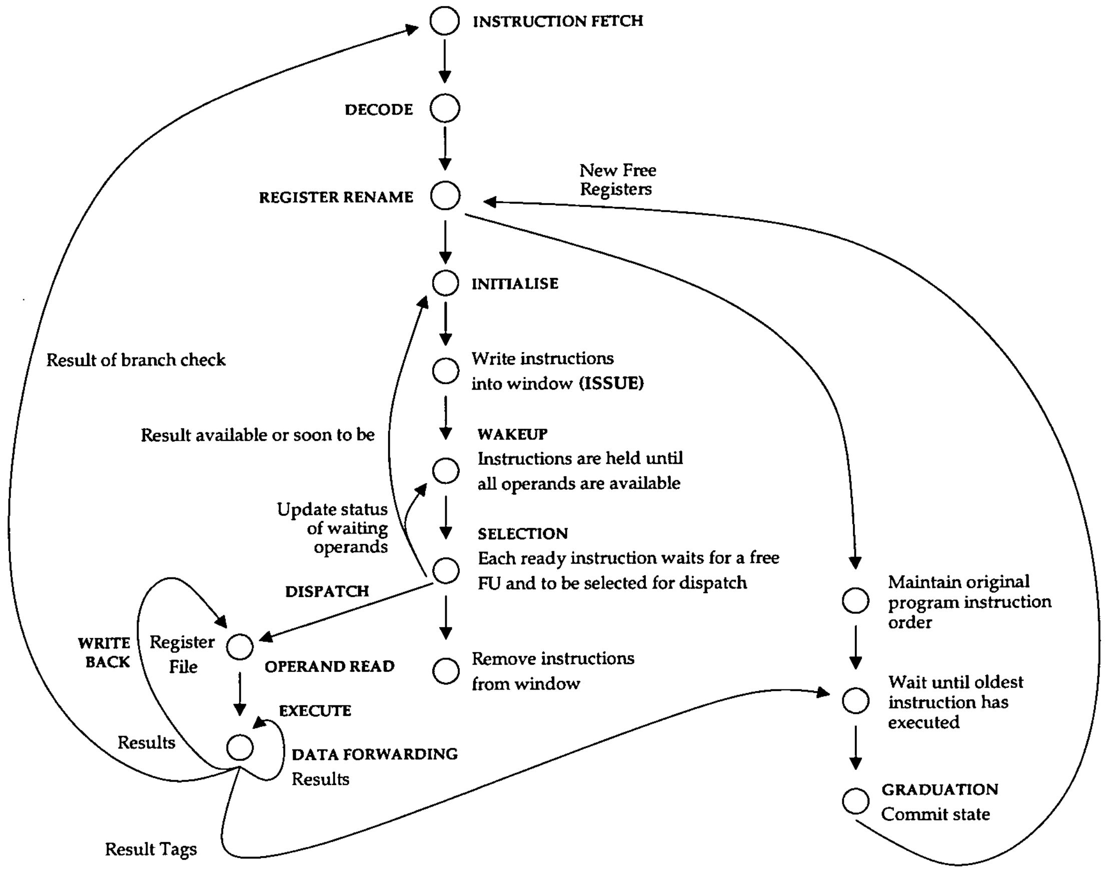

INSTRUCTION FETCH
DECODE
REGISTER RENAME
New Free Registers
INITIALISE
Result of branch check
Write instructions into window (ISSUE)
Result available or soon to be
WAKEUP
Instructions are held until all operands are available
Update status of waiting operands
SELECTION
Each ready instruction waits for a free FU and to be selected for dispatch
DISPATCH
Maintain original program instruction order
WRITE BACK
Register File
OPERAND READ
Remove instructions from window
EXECUTE
Wait until oldest instruction has executed
Results
DATA FORWARDING
Results
GRADUATION
Commit state
Result Tags

Figure 3.1: Simplified view of instruction and data-flow in a generic superscalar processor

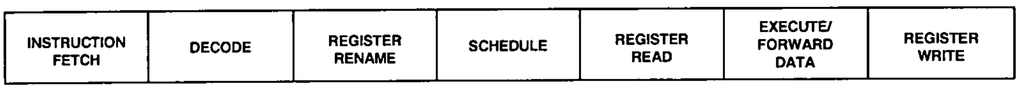


Figure 3.2: Pipeline stages of a generic superscalar processor

The remaining communications: to add new registers to the free register pool and communicating the outcome of branches to the instruction fetch stage are effectively decoupled, and as such do not represent a synchronisation problem. The free register pool may be implemented as a FIFO, only forcing synchronisation between the renaming stage and the reorder buffer if the free register FIFO becomes empty. In practice, sufficient registers may be provided for this never to occur. The instruction fetch stage decouples itself by speculating on the outcome of branches.

While other architectures could have been taken as a starting point, this particular organisation creates a simple interface between the scheduling and execute stages, and the rest of the processor. This provides a good starting point for investigating a range of asynchronous designs, without being disadvantageous to any synchronous implementation. The following sections describe new approaches to both dynamic scheduling and data-forwarding which operate without the need to be performed in a pseudo-synchronous fashion.

3.3 An asynchronous dispatch buffer

The dispatch buffer creates a window into the dynamic instruction stream with the ability to dispatch[1] any instruction, irrespective of its age, when its operands and a free FU become available. A clear requirement of any asynchronous implementation, in addition to minimising synchronisation, is to provide a low-latency implementation of both the wakeup and selection processes.

A block diagram of the register rename and dispatch buffer pipeline stages is shown in Figure 3.3. The busy-bit table is accessed by both the register rename stage and dispatch buffer. The table contains an entry for each physical register, consisting of a single bit indicating the status of the instruction that will write to it. During register rename each instruction resets the entry in the busy-bit table corresponding to its physical destination register. The entry is set once the instruction is dispatched. The table provides the first of two mechanisms used to update the status of instruction operands. Instructions in the register rename stage query the table prior to being written into the buffer, which provides the initial state of their operands. The second mechanism is the wakeup operation performed when an instruction is dispatched, which updates the status of any of the operands in the buffer that match the dispatched instruction's destination register identifier.

At least two possible implementations exist for the underlying instruction buffer, these include: the micropipeline introduced in Section 2.4.3 and the parallel FIFO [161, 160, 17]. The micropipeline operates by propagating entries from its input to its output along a linear array of identical stages. In contrast, the parallel FIFO implements a hardware version of the circular array buffer. During a write or read, the appropriate buffer entry is indicated by a write or read pointer and accessed directly. The use of a parallel FIFO allows new instructions to be written directly into a fixed buffer memory location, where they remain until they are removed. This simplifies the implementation of both wakeup and selection operations, as additional arbitration between these processes and the propagation of entries may be omitted. While we discuss only implementations

[1]The word *issue* is used to describe the movement of an instruction from the decode stage of a pipeline to an instruction or dispatch buffer. The word *dispatch* describes an instruction leaving this buffer and being sent to a functional unit.

based around a parallel FIFO, designs utilising micropipelines in a counterflow organisation are also possible [155, 154] and are discussed later in this chapter. One possible benefit of such an approach is that compaction of the buffer is obtained at a low cost, with empty slots in the buffer being filled as instructions propagate. However, the inability to perform a parallel wakeup operation is likely to severely limit performance.

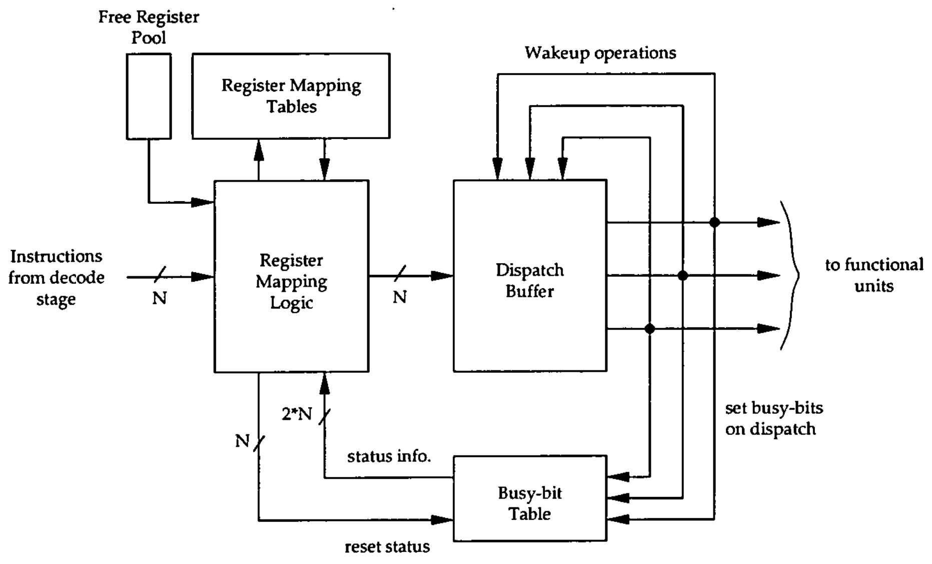


Figure 3.3: A block diagram of the register rename and dispatch buffer pipeline stages

3.3.1 Communicating with the dispatch buffer

In the following section we describe how access to the dispatch buffer is controlled. The goal is to ensure correct operation while also maximising concurrent access to the buffer.

To guarantee that all instructions are eventually dispatched we must ensure that the status of every operand is updated. In the organisation shown in Figure 3.3, the possibility exists for an operand to read a busy-bit entry indicating that its parent instruction has not been dispatched, while also missing the same instruction's wakeup operation. This may occur as the initialisation and the buffer write processes operate completely asynchronously to the dispatch/wakeup processes. This creates the possibility that a wakeup operation and a write may occur simultaneously. While this behaviour is acceptable when the register identifiers of the operands being written and the wakeup process differ, we cannot guarantee a match and successful updating of the operand's status if they are the same.

The problem is solved by arbitrating access to the busy-bit table and dispatch buffer. The simplest scheme would involve the register rename stage and wakeup processes requesting access to the dispatch buffer and busy-bit table through a single arbiter. This first scenario is illustrated in Figure 3.4(a) in the form a petri-net. Only one of the processes is permitted to acquire the token from the mutual-exclusion place at any one time. The token is only returned allowing another process to proceed, when the process that receives the token has completed. This scheme, while providing a correct solution, sequentialises all of the initialise and wakeup operations that are performed on the window. One possibility would be to stall wakeup until a number of wakeup operations could be performed in parallel. In practice, as the wakeup processes are asynchronous this is difficult to achieve. Stalling wakeup operations will also quickly lead to worst-case behaviour.

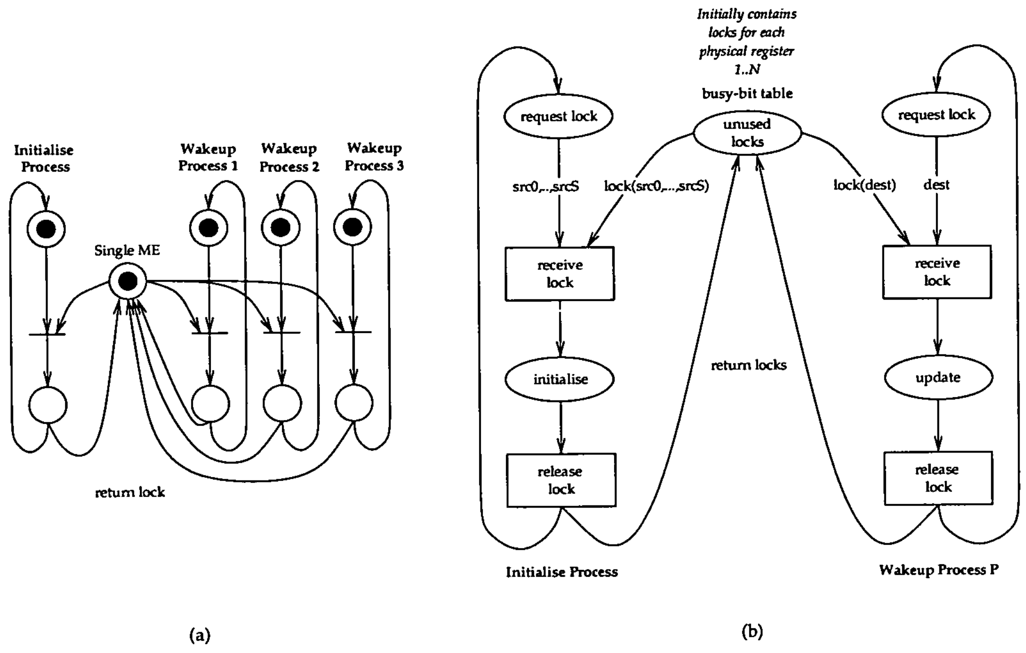


Figure 3.4: Petri-net representations of two schemes for enforcing mutual-exclusive access to busy-bits and dispatch buffer entries.

Synchronisation may be minimised in this case by enforcing mutual-exclusion only when absolutely necessary. As described previously, permitting concurrent access to the busy-bits and dispatch buffer is only problematic when the initialisation process and wakeup processes access the same busy-bit. This occurs when the update process is attempting to modify the status of an operand that is required by one or more of the instructions in the register rename stage. By providing an arbiter at each entry in the busy-bit table correct operation may be guaranteed, while allowing maximum concurrency between each of the processes. This organisation is illustrated in the second Petri-net (Figure 3.4(b)). A wakeup process now only requires the token from the busy-bit place which corresponds to its destination register identifier. Similarly, the initialisation process only requires tokens for each of the instruction operands currently in the rename stage. Each process now performs the following operations in sequence: obtains the necessary lock or locks, operates on the busy-bits and dispatch buffer and then returns the lock(s). A detailed description of how this mechanism may be implemented is provided below.

A possible implementation of the arbitration and state-holding logic required within each entry of the busy-bit table is shown in Figure 3.5. Together with the decode logic necessary to steer read and write requests to each of the entries, this forms both the data-dependent arbitration scheme and busy-bit table.

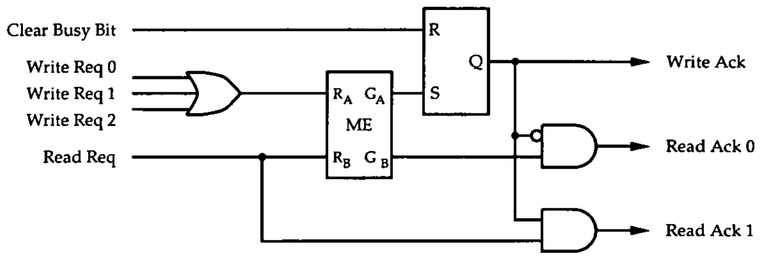


Figure 3.5: Arbitrated Access to an Individual Busy-Bit Table Entry

The SR-Latch stores the dispatch status of the instruction corresponding to the busy-bit entry. The latch is initially cleared when a new physical register is assigned to an instruction. The task of clearing a particular busy-bit may actually take place at any time between the point at which the corresponding physical register is reclaimed and added to the free register pool, and the point at which it is allocated to a new instruction.

Register renaming guarantees that no two instructions with the same physical destination register may be in flight simultaneously. This guarantees that no two wakeup processes will ever attempt to access the same busy-bit table entry simultaneously. A number of possible sources of write request are therefore combined using an OR gate. Write requests are generated by wakeup processes and set the status of the busy-bit to indicate that the instruction corresponding to the entry has been dispatched. The operation is acknowledged as soon as the request is granted by the mutual exclusion element and the busy-bit has been set. Register renaming will of course guarantee that no further write requests are generated until after the entry has been cleared.

Read requests are generated by the initialisation process to obtain the initial status of instruction operands. Initialisation occurs during the register rename pipeline stage after physical register identifiers have been obtained. In practice, dependencies within the group of instructions fetched in parallel may be determined without reference to the busy-bit table. This reduces accesses to the busy-bit table and permits all busy-bit table entries associated with the destination registers of the instructions in the group to be reset simultaneously. A further optimisation may be made by allowing busy-bit entries that are set to be read without the need for the arbiter to provide a read grant. This is permissible as we can assume that if the entry is set, no further write operations can take place. This functionality is provided by the two AND gates. The upper gate prevents the `Ack 0` output from going high in this case, while the lower gate allows the mutual exclusion element to be bypassed.

3.3.2 Selecting ready instructions for dispatch

The previous section described a scheme that minimised the synchronisation required between the initialise and wakeup processes. This ensured that we maintained the correct status of instruction operands waiting in the dispatch buffer. The final requirement is to provide a scheme that selects an instruction for dispatch. Synchronous solutions are simplified by the fact that the status of instructions does not change during the selection process. In an asynchronous implementation however, where the aim is to minimise synchronisation, wakeup operations occur asynchronously and may produce ready instructions at any time.

The implementation of the selection logic has a number of potential implementations. Conceptually, the simplest is to construct an n-input asynchronous arbiter. Large fan-in arbiters may be constructed as tree arbiters [69], meshes or token rings [158]. To avoid the dispatch stage becoming a bottleneck it is important that the latency of such an arbiter is minimised. Studies have shown that the performance impact of the selection policy itself is small [24], *e.g* selecting older ready instructions first or selecting ready instructions at random. Hence, an approach that minimises latency at the expense of fairness [26] and the ability to prioritise grants is acceptable. We next describe two possible solutions.

The first one is to use a tree arbiter where a large multi-way arbiter is constructed as a tree of two-input tree-arbiter elements. When a request is first made at one of the inputs to the arbiter, the request signal that travels up the tree is generated by a simple combinational logic function at each element. This allows the request to travel quickly up the tree and hides much of the delay of the slower mutual-exclusion elements, which grant concurrently with the propagation of the request. This scheme is illustrated by considering the highlighted path in Figure 3.6. Here it is clear that the request `r9` may be generated before it is determined that input, r5 or r7 will receive the final grant. The result of the arbitration required to determine that one will be granted will only be required when the grant signal g9 is raised. Two recent designs that employ this type of eager request propagation and also allow eager acknowledge of releases are presented in [159, 69].

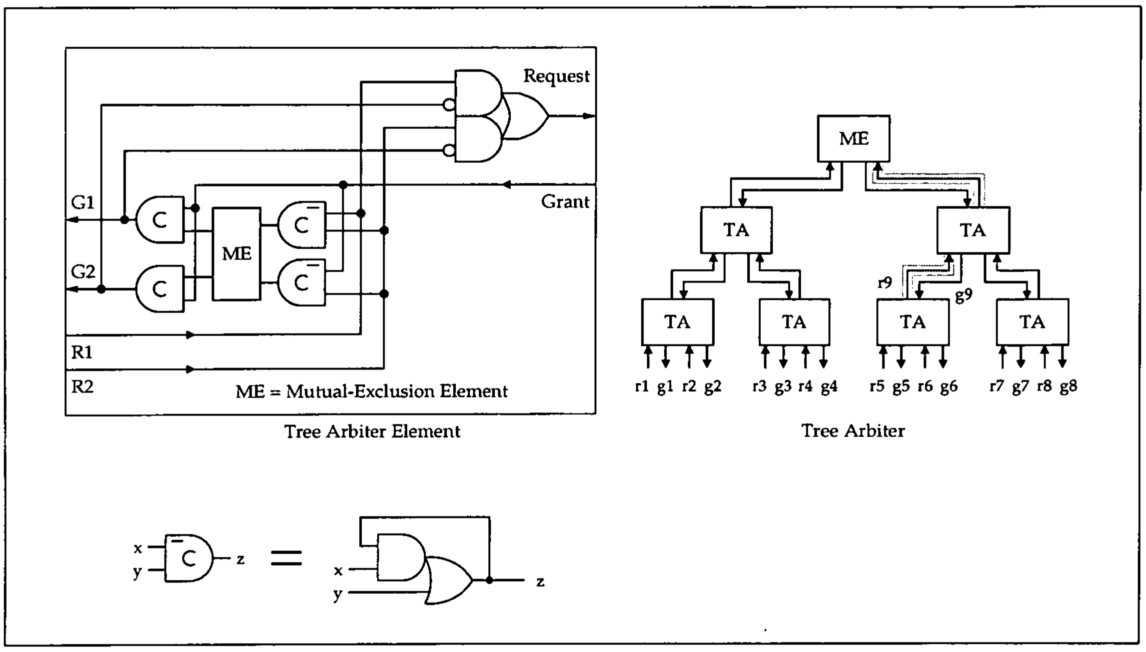


Figure 3.6: Tree arbiter element and multi-way arbiter block diagram

Figure 3.6 also provides a possible implementation of a tree-arbiter element. The design differs slightly from that presented in [69]; for the sake of simplicity the design here has been mapped to standard cells (and a ME element). The possibility of skewing the tree to favour of particular inputs, for example those requests from instructions at the head of the dispatch buffer, is unfortunately unrealistic as the location of the head entry in a parallel FIFO changes.

While tree arbiters potentially offer good performance by exploiting concurrency in their operation, an alternative solution is to provide static inputs to a purely combinational selection function. Such a scheme attempts to mimic the synchronous case where an instruction's status cannot change during selection. This offers the possibility of simplifying and reducing the latency of the selection process. Such a selection scheme may be implemented in the following way:

1. One or more instructions are detected as being ready. This `any ready` signal is simply the logical OR of the `ready` outputs generated by each reservation station.

2. The detection of a ready instruction prompts the selection logic to request that each reservation station produces a `ready` or `not ready` output. This output is fixed until the removal of the selection logic's request signal.

3. The selection logic may now be simplified with the knowledge that the status of instructions will not change during selection.

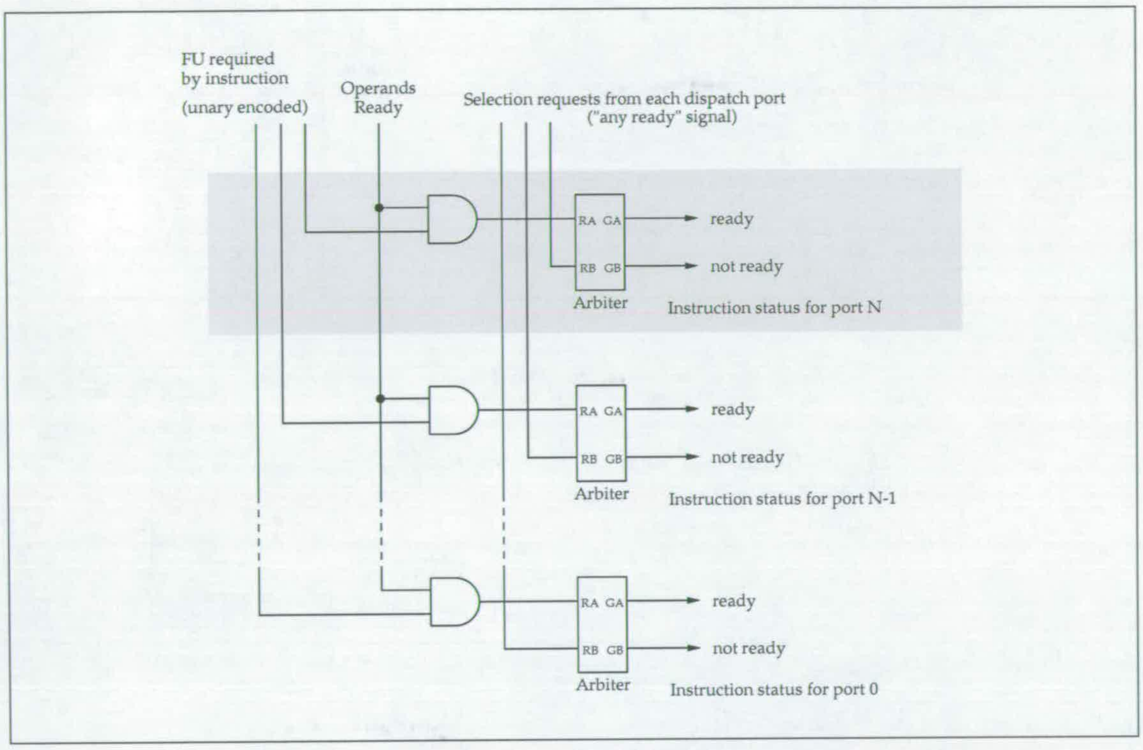

FU required by instruction (unary encoded)
Operands Ready
Selection requests from each dispatch port ("any ready" signal)
RA GA
RB GB
Arbiter
ready
not ready
Instruction status for port N
Instruction status for port N-1
Instruction status for port 0

Figure 3.7: Selection Arbiters for a single reservation station. Highlighted area shows the logic corresponding to a single FU dispatch port.

We assume at this point that there is a dispatch process associated with each functional unit. To ensure maximum concurrency and minimum synchronisation between each of the dispatch processes, we must implement selection logic independently for each dispatch port. The logic required to create the outputs, as described in step 2 above, is shown in Figure 3.7. Here an arbiter is required for each dispatch port to provide a decision on whether a `ready` or `not ready` signal is established prior to selection. For a particular dispatch port only those ready instructions requiring the functional unit associated with the port will output a ready signal.

A block diagram of the selection logic is shown in Figure 3.8. A similar structure is used in the PA-8000 processor [79] and is also described in [108]. The selection logic operates in a way similar to the tree arbiter described previously, although its implementation is simplified by removing the need for mutual-exclusion elements at each node.

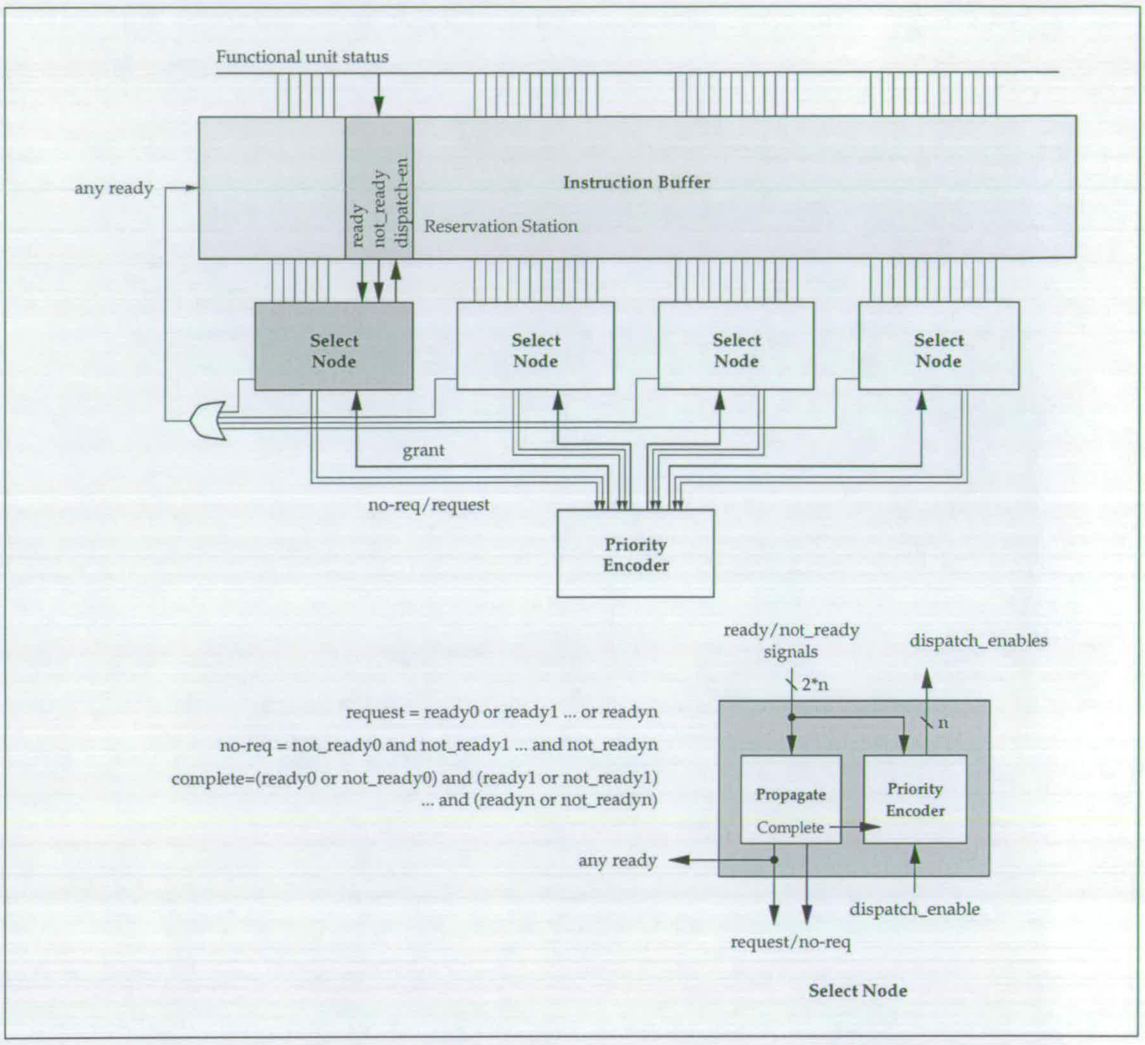


Figure 3.8: Selection logic for an asynchronous dispatch buffer.

The selection nodes create two outputs in response to the `ready` and `not-ready` inputs from the reservation stations. Firstly, an eager `request` signal is generated as soon as any one of the `ready` inputs is raised. This `any-ready` signal is combined with the outputs of other select nodes and broadcast to all entries in the dispatch buffer. This eventually forces every reservation station to produce a `ready` or `not-ready` output. Concurrently, each `ready` signal is propagated to the next level of the tree or the root node priority encoder. If none of the `ready` inputs to a select node are set, then the node will wait until all the `not-ready` inputs are available before raising the second output (`no-request`). This signifies that all reservation stations have responded but none require a `dispatch enable` signal.

As soon as the root node priority encoder has a valid input from each select node (`no-request` or `request`) it is able to grant one of the nodes that requires a `dispatch enable`. This takes place concurrently with the completion detection and priority encoding that is necessary in each select node. Within a node, the priority logic is stalled until a valid input is detected from each reservation station and the `complete` signal is raised. The subsequent receipt of a `dispatch enable` allows the ready entry with the highest priority to be enabled for dispatch.

Further optimisations may be sought by observing that the `any-ready` signal may stay high between successive selection operations if a number of ready instructions are detected. Schemes could also focus on exploiting opportunities to enable early dispatch under some conditions. For example, enabling dispatch before the `complete` signal is raised if the highest priority `ready` input is set. In practice, in the case of the selection logic the aim should be to ensure a low worst-case latency, which will ensure that the selection logic rarely, if ever, limits performance.

However when compared to a typical synchronous implementation of selection logic a small latency overhead may be seen. This would probably at most be the time required to detect one ready entry, together the delay of the arbiter required to generate `ready` and `not-ready` outputs at each reservation station. In general, the approach provides a low-latency alternative to the tree-arbiter approach while introducing no additional synchronisation or arbitration requirements. The minimisation of the worst-case selection latency is important as it is exposed when attempting to execute dependent instructions in succession. Introducing an additional latency between two such instructions immediately limits the levels of ILP that may be exploited.

Independent of which implementation style is adopted, further reductions in latency may be possible by assigning instances of duplicated FU units to different subsets of the dispatch buffer. For example, assigning different ALUs to odd and even buffer entries. The resulting decrease in the size of the selection problem provides a corresponding reduction in latency.

3.3.3 Summary

The previous sections have described techniques for implementing a completely asynchronous dispatch buffer. Synchronisation between the processes that access the buffer is only enforced to guarantee correct operation. This requires that initialise and wakeup operations that reference the same result do not access the dispatch buffer and busy-bits simultaneously. Furthermore, a realistic implementation of the selection logic is provided, again without the need to resort to synchronising the operation of the dispatch or wakeup processes.

At this point it is useful to contrast the operation of the original synchronous buffer and the new asynchronous implementation. In the original synchronous buffer the wakeup and selection phases of dynamic scheduling are performed strictly sequentially, while in the asynchronous scheme they operate concurrently. Of course, the dispatch of ready instructions in the synchronous scheme is also synchronised. The asynchronous implementation provides fully independent dispatch processes, allowing instructions to be dispatched at any time. The need to support both concurrent wakeup and initialisation processes is critical when we are attempting to exploit the actual delays exposed in the following pipeline stages. Another consequence of operating wakeup and selection processes independently is that ready instructions may potentially be dispatched faster than the synchronous rate of one per clock cycle.

3.4 Data Forwarding

In a synchronous processor, the execution of data-dependent instructions in consecutive clock cycles requires that the result of the first instruction be communicated to the second as soon as possible. The delay incurred in the execute pipeline stage of a synchronous processor includes both this bypass delay and the delay of the functional unit itself.

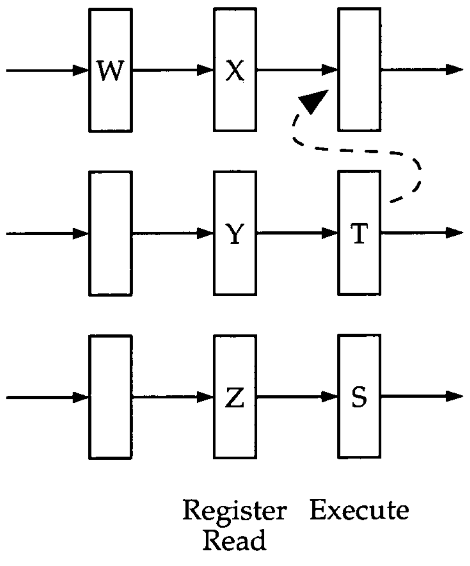


Figure 3.9: Example of forwarding between parallel execution pipelines

Figure 3.9 shows three parallel instruction pipelines, the dotted line represents a communication between the top two pipelines. This data-forwarding operation supplies instruction X with the result of instruction T. In a synchronous processor the detection of such an opportunity to bypass the register file is easy, as all pipelines progress in lockstep. When instruction X enters its operand fetch stage a comparison between its operands and the destination register identifiers of the instructions currently in the execution stage determines if forwarding can take place. The result generated by T can then be obtained from the appropriate bypass at the beginning of the next cycle.

In an asynchronous processor we wish to minimise synchronisation between the pipelines in order to exploit data-dependent delays and minimise control overheads. This requires a data forwarding mechanism that does not rely on the lockstep operation of instruction pipelines. We must also be conscious of the performance implications of increasing operand fetch latency, either by reducing opportunities to forward data or by increasing the latency of the data-forwarding operation itself. One potential advantage of any asynchronous implementation is that, if forwarding is not required or can be initiated before an instruction enters its operand fetch stage, then there is scope for improving average performance.

The following sections first describe how data-forwarding may be implemented in the simple case where prior knowledge about the dispatch order of instructions may be exploited. The scheme is then extended to support the asynchronous dispatch buffer discussed in the previous section.

3.4.1 Data forwarding with locally in-order dispatch

Figure 3.10 shows a scheme for implementing a limited form of dynamic scheduling. The complex dispatch buffer is replaced with a number of instruction queues. Each queue dispatches instructions to its local functional unit in the order they were issued to it. While the instruction schedule is fixed for each functional unit, the rate at which each instruction pipeline proceeds may vary. Queues only become synchronised when data-dependencies exist between queues, potentially forcing one queue to stall until a result is available.

The assumption that instructions are executed at each functional unit in the same order as they are issued may be exploited to detect opportunities to forward data. A similar approach is exploited in the Hades and AMULET2 architectures described in Section 2.6. For the sake of simplicity, we first describe a scheme where only results from the last instruction issued to each dispatch queue are considered for forwarding. A more general scheme is introduced later.

Figure 3.11 illustrates the steps required for forwarding data in such an architecture. The issue unit holds the destination register identifiers for the last

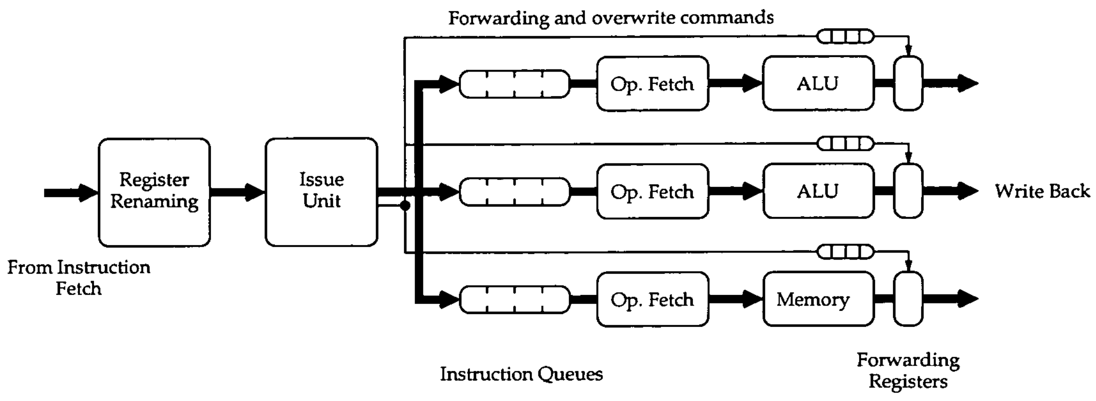


Figure 3.10: A simple queue-based dynamic scheduling scheme. The operand fetch stages may read data from either the register file (RF) or forwarded data result queues (the RF and buses associated with operand fetch are omitted for clarity).

instruction issued to each instruction queue. The current values for each step in the example are listed underneath the Issue Unit (labeled Forwarding Table). Opportunities to forward data are detected in the Issue Unit by comparing instruction operands to those register identifiers stored in the forwarding table. Forwarding registers temporarily hold results at the output of each FU. Data may be written to, or forwarded from, these registers under the control of the issue unit.

Step 1 in the example, shows the state of the datapath just after the first two instructions from the program fragment have been issued. The issue unit also sends accompanying overwrite commands. When these commands reach the head of the forwarding register command queue, then they allow new data to be written into the forwarding register.

At the beginning of step 2, a comparison between the operands of instruction $I3$ and those register identifiers stored in the Issue Unit results in two matches. In response, the issue unit creates two forwarding requests: one to $FU0$ and one to $FU1$, each request indicates that the result held in the forwarding register when the request is received should be sent to $FU2$.

Step 3, illustrates the point at which $I1$ has created a result, and the forwarding request has been received at the forwarding register. The forwarding command is executed and the data in the forwarding register is sent to instruction $I3$ in $FU2$'s operand fetch stage. In addition to the forwarding operation, a new instruction ($I4$) is shown in the Issue Unit. As this instruction requires $FU0$, the Issue Unit sends an overwrite command to clear the contents of the forwarding register. The register identifier of the new instruction is also noted, replacing that of $I1$.

An obvious problem with such a scheme is that the order in which data is forwarded to a particular functional unit from different pipelines cannot be guaranteed. For example, if two instructions at $FU2$ request data from $FU0$ and $FU1$ respectively, then the order in which the results arrive at $FU2$'s operand fetch stage is non-deterministic. This problem could be overcome by ensuring that we only forward data when the order of the arrival of forwarded results at a particular FU could be guaranteed. A particular order could be the result of other data dependencies or additional hardware introduced to guarantee the ordering of results. Such approaches would most likely either reduce opportunities for forwarding, and increase synchronisation or add to the latency of the forwarding operation.

A general solution to the problem is to provide a buffer for each possible source of forwarded data at each operand fetch stage. An instruction entering its operand fetch stage is able to select data from the correct queue by examining a tag added to each operand during issue. The tag indicates whether forwarding is possible, and if so, the source of the data. A block diagram illustrating a single execute pipeline incorporating data forwarding is shown in Figure 3.12. The technique works as we can assume that results from a particular FU will always arrive in the order in which they are sent.

One final requirement is to guarantee only a single FU source is recorded for each result in the Issue Unit's Forwarding Table. For example, consider the case where three instructions with the same destination register are issued consecutively – this would result in three different sources being recorded for the same register. The problem can be avoided by first deleting those table entries that match a newly-fetched instruction's destination register. In practice this operation may be performed each time a new register is added to the free register pool, preventing any lengthening of the issue stages cycle time. Correct operation is ensured by only allocating registers to new instructions after their entries have been deleted from the forwarding table.

The buffering of forwarded data is also beneficial to prevent unnecessary pipeline stalls. In a synchronous architecture the forwarding operation always takes place just prior to the use of the result. In the asynchronous queue-based architecture, a forwarding request may be received and serviced well before the data is actually required. The provision of buffers allows such forwarding operations to take place while preventing the need to stall the source pipeline until the consumer instruction has reached its operand fetch stage.

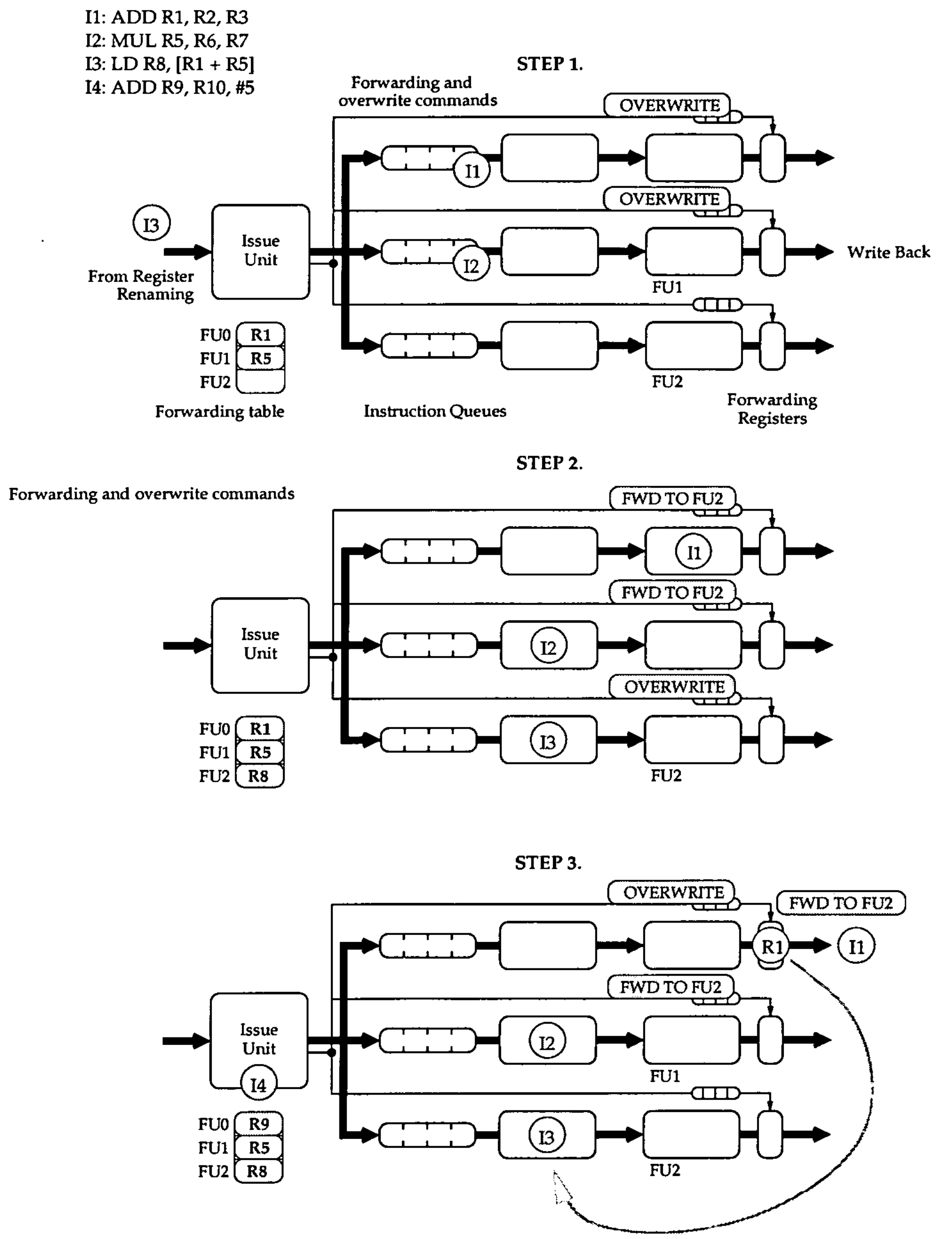


Figure 3.11: Step-by-step illustration of a simple asynchronous forwarding mechanism

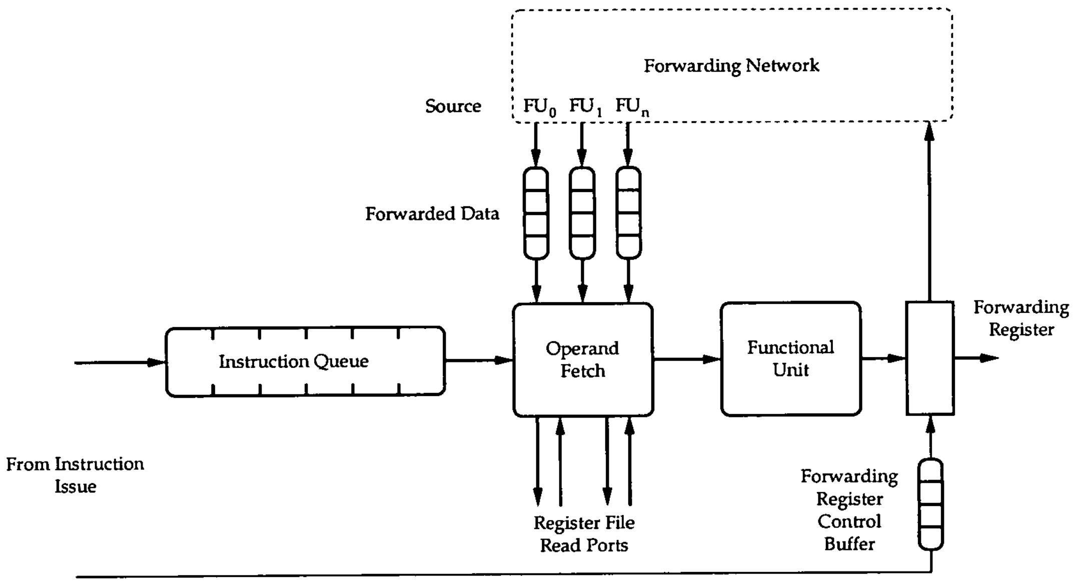


Figure 3.12: Execution pipeline with Data Forwarding

It should be noted that data forwarding may be preferable to obtaining operands from the register file from a power perspective even when performance is unaffected. Reductions in register file traffic may provide an opportunity to reduce the total number of read ports, leading to a decreases in their access time. The forwarding operation is also likely to consume less energy. The levels of forwarding may be increased by providing more forwarding registers at each FU. This allows the results from the previous N instructions issued to each instruction queue to be considered for forwarding during issue.

The first step is to increase the number of forwarding registers at the output of each functional unit to N. The forwarding register command queues remain, while the forwarding registers themselves are organised as a circular buffer. The overwrite command now removes the oldest entry from the FIFO, allowing a new result to be written in its place - of course this can only be permitted to take place after the old result has been written to the register file. The forwarding table in the issue unit is now expanded to store the destination registers of the previous N instructions issued to each instruction queue. The number of operand/destination register comparators is also increased to allow all the table entries to be searched in parallel. Forwarding requests are now made as before, but are tagged with the particular entry in the buffer that is required. The implementation of the circular buffer would be very similar to the parallel FIFO described earlier.

3.4.2 A simple write-back scheme

The forwarding scheme described previously is suitable for inclusion in a simple multiple-issue architecture; one in which scheduling at compile-time is more critical than in a typical superscalar machine. To reduce the complexity of such an architecture, the register renaming and reorder buffer hardware may be removed and replaced with a simpler scheme to support precise interrupts and speculation. For example, a result shift register scheme similar to the one used by the MiniMIPS architecture may be used (see Section 2.6.5). This ensures that instructions only ever write to the register file in program order, removing the need for a reorder buffer. Unlike the MiniMIPS architecture, access to results generated out of order would be provided through the use of the forwarding mechanism described in the previous section. This type of architecture is illustrated in Figure 3.13. The approach can be thought of as a distributed version of the AMULET3's reorder buffer. The reorder buffer provides a slightly more general forwarding scheme enabling results of the previous N instructions to be forwarded, irrespective of the FU they use. More important than the generality of the distributed forwarding scheme, is its ability to buffer results locally in order to minimise pipeline synchronisation. A distributed implementation of forwarding is also preferable in order to minimise the latency of the forwarding operations.

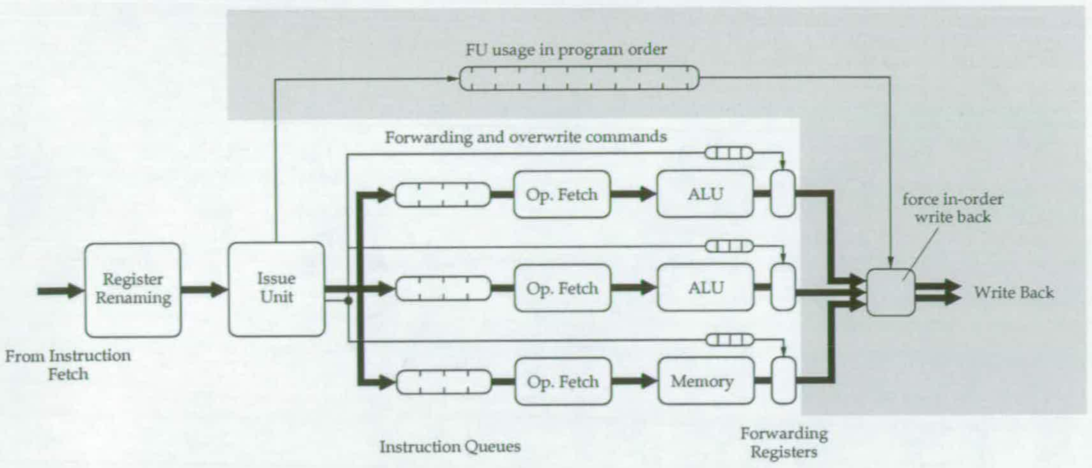


Figure 3.13: The queue-based processor with additional hardware to enforce in-order write-back (highlighted). The operand fetch stages may read data from either the register file (RF) or forwarded data result queues (the RF and buses associated with operand fetch are omitted for clarity).

If the table-based register renaming mechanism was also removed, we could no longer assume that each instruction is assigned a unique destination register. This forces instructions to stall during issue if they detect that their destination register is already locked.

3.4.3 Data forwarding with out-of-order dispatch

We now describe how data forwarding, as detailed in the previous sections, may be employed when instructions are dispatched out of order from a central dispatch buffer.

Consider the organisation shown in Figure 3.14. In contrast to the queue-based architecture, overwrite and forwarding requests can now only be made once an instruction reaches its operand fetch stage. This is because the order in which instructions will be dispatched is not known. Potential sources of data are now identified dynamically during wakeup operations. For example, if an instruction is dispatched to the memory unit, then any operands requiring its result will record the memory unit as the source of data. Wakeup operations that match particular operands and set their status to ready, now also write this source FU data into the reservation station. Instruction operands that are initialised to be ready at the time the scoreboard is read will not generate forwarding requests. In reality such data will often already be available from the register file. To summarise, a forwarding operation now involves the following steps:

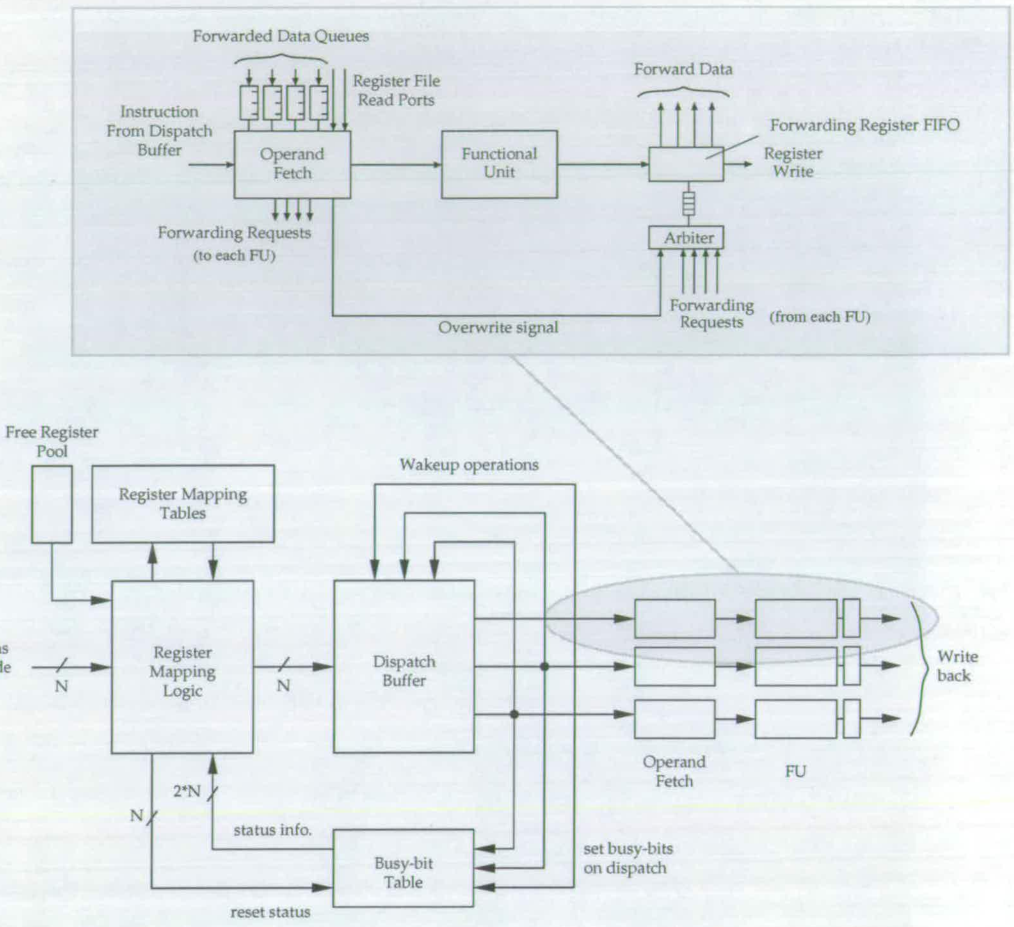


Figure 3.14: Data Forwarding supporting Out-of-order Dispatch

1. Wakeup processes broadcast the destination register identifiers of dispatched instructions to all entries in the dispatch buffer.
2. The status of those operands that have the same register identifier as the broadcast result are set to ready. The FU at which the result is generated is also appended to the instruction in the same reservation station.
3. Instructions are dispatched when all operands are ready and a suitable FU is free.
4. Forwarding requests are made, if necessary, for each operand. Operands that cannot be obtained via forwarding are fetched from the register file.
5. An overwrite signal is sent to the forwarding registers at the output of the instruction's functional unit. This invalidates the contents of the oldest forwarding register in preparation for a new result to be latched.

At this point it is clear that concurrent forwarding requests may be made from a number of operand fetch stages to a single forwarding register control buffer. One problem is that overwrite signals, invalidating the current contents of a forwarding register, may also be sent concurrently with forwarding requests for the current data. In the queue-based architecture this problem was avoided by ordering overwrite and forwarding requests from a centralised issue unit. When dispatch is truly out of order, such requests must be sequentialised through arbitration. This arbiter component is shown in Figure 3.14 (just below the forwarding buffer).

Given that requests are now generated in a distributed fashion, the forwarding requests may be received too early or too late; either prior to the receipt of an overwrite signal preparing for the receipt of a particular result, or after subsequent overwrite signals have effectively removed the data. To avoid the possibility that incorrect data is forwarded, the forwarding requests must now include the register identifier of the required result. A comparison in the forwarding buffer now determines if forwarding is possible, or if the forwarded operation must be cancelled. In the latter case, a signal is sent to indicate that the data is not available, which forces the result to be obtained via the register bank.

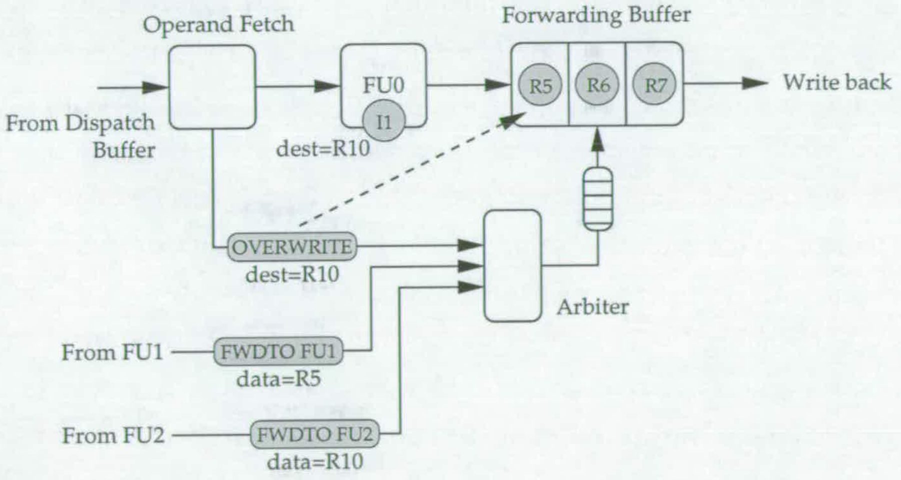


Figure 3.15: An example of how forwarding requests cannot always be guaranteed to be successful if out-of-order dispatch is employed

The situations leading to successful and unsuccessful forwarding requests will now be illustrated using Figure 3.15. Instruction `I1` will generate a result destined for register `R10`. To enable the result to be written into the forwarding buffer, and to make it available for forwarding, `I1` generates an overwrite command when in its operand fetch stage. When this command reaches the forwarding register, the entry currently holding the contents of `R5` will be overwritten. The success of the two forwarding requests from FU1 and FU2, for results R5 and R10, respectively, now depends on the order in which the forwarding request and overwrite commands are granted:

- If the overwrite command is granted first, then the contents of R5 will no longer be available for forwarding. When the forwarding request from FU1 for R5 reaches the forwarding registers, a response indicating that forwarding was not possible will be sent in place of the result data. When the second forwarding request for R10 reaches the forwarding registers, the register identifiers will match and forwarding may take place. In practice, the forwarding operation may stall until the data has actually been generated and written to the register.

- If both forwarding requests are granted before the overwrite command, then the forwarding request for FU1 will be successful, while the request for R10 will fail as no overwrite command has yet been received to initialise its entry in the forwarding buffer.

The point at which the overwrite signal is generated defines the precise characteristics of this window in which a forwarding request will be successful. In the extreme case, the overwrite command could travel with the instruction through the functional unit, only arbitrating with forwarding requests after the result has been generated. Such a scheme would deny all forwarding requests made for the result of an instruction that has not generated a result - even if the request was made just prior to a result becoming available. In the other extreme, generating overwrite commands too early is likely to reduce the effectiveness of the forwarding registers, reducing the chance of forwarding data from the oldest entry. Our simulations have showed that a reasonable trade-off may be made by generating overwrite signals during operand fetch after an instruction has received its operands.

3.4.3.1 Reducing the number of unsuccessful forwarding requests

In the scheme described previously it would be beneficial if it could be predicted when forwarding requests were likely to be unsuccessful. This would allow the total number of forwarding requests to be reduced, while having no impact on the actual number of results forwarded. Such a reduction in surplus forwarding requests would in many cases decrease the time required to service those that remained. The mechanism described next operates by resetting forwarding operations before an instruction is dispatched, but some time after forwarding has been initialised.

To enable forwarding operations to be reset, a *source counter* is associated with each operand in the dispatch buffer. When a wakeup operation matches an operand, then its status is set to ready and the source of the result is noted. At the same time the source counter is initialised. Each subsequent wakeup operation from the same functional unit now reduces the value of the source counter (irrespective of the destination register identifier of the result). If the counter is reduced to zero, then the source of the data is reset to the register file and a forwarding request will no longer be made. The counters simply predict the case when it is likely that the contents of the forwarding register have been overwritten by subsequent dispatches to the same functional unit.

Counters in the dispatch buffer are only updated after forwarding has been initialised. This removes any concern about the behaviour of the counters when a new instruction is written to the window. To guarantee that a counter is not modified during dispatch, an additional arbiter is required within each reservation station. This ensures that counter decrement and dispatch operations are mutually exclusive.

3.5 Alternative approaches

The following sections explore two existing designs for asynchronous ILP architectures.

3.5.1 Counterflow-pipeline based dispatch buffer

In [155] Werner and Akella describe an asynchronous implementation of the Fast Dispatch Stack (FDS) [2, 37]. The dispatch stack architecture exploits a large amount of unary encoded data to minimise the critical path of the scheduling hardware. The motivation for an asynchronous version came from the idea that a counterflow pipeline could be used to simplify the compaction and dependency resolution logic. Dependency resolution is complicated by the fact that no register renaming takes place, which requires that both true- and false-dependencies be detected.

A block diagram of the architecture is shown in Figure 3.16. The buffer unit prepares instructions for entry into the issue unit, it is here that a number of unary-encoded vectors are appended to the instruction. These vectors simplify the process of detecting dependencies with instructions waiting to be issued higher in the issue unit's instruction pipeline.

Read (source) and write (destination) vectors identify the registers used by each instruction. In addition a unique result tag, again unary encoded, is associated with each instruction. These vectors together with the instruction form an I-group. I-groups flow from the bottom of the issue unit towards to the top. The other pipeline, which flows in the opposite direction carries dependency information. The pipelines operate together as a counterflow pipeline.

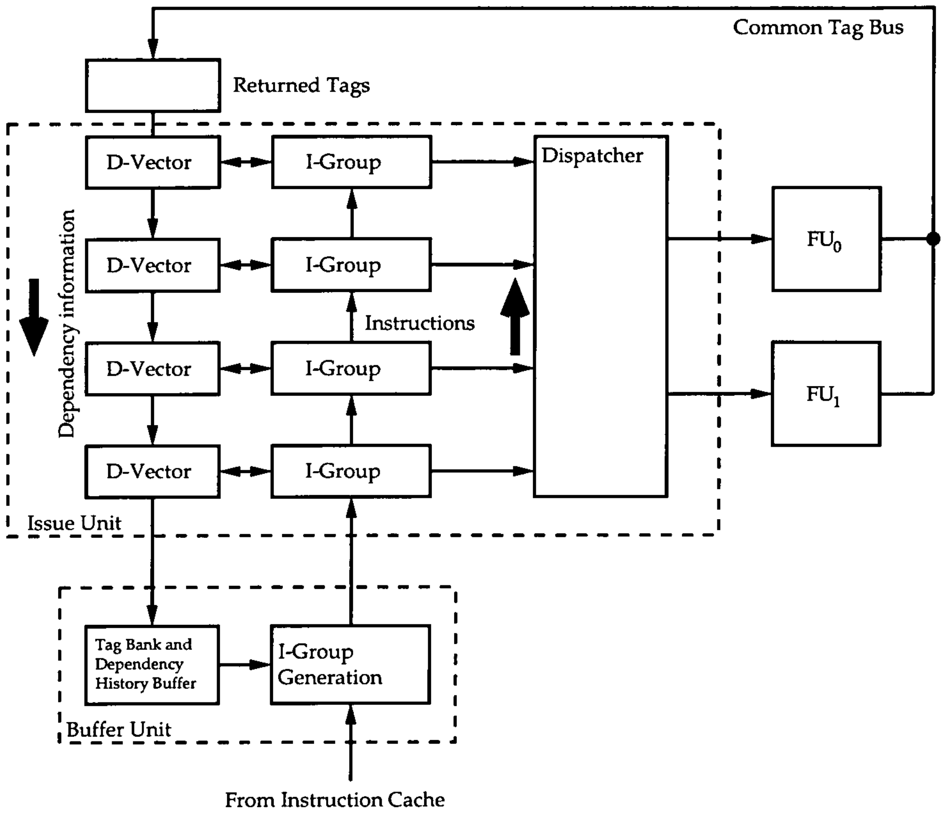


Figure 3.16: AFDS Block Diagram

Tag vectors are issued with an instruction to its functional unit; on completion, the tag is returned to the issue unit and enters the dependency pipeline at the top. The dependency pipeline contains two vectors – read and write. These vectors incorporate dependency information as they pass each I-group. At any particular stage in the pipeline they allow all potential dependencies created by the instructions that they have passed to be represented. Each instruction uses the information provided in the dependency pipeline to determine if it may issue. The result tag is never modified and is simply used to remove instructions from the issue unit that have completed; the so-called evaporation takes place whenever the result tag matches the tag contained within the I-group.

The selection process is controlled by the instruction dispatcher. A call request from the dispatcher queries each instruction in the buffer, forcing an issue request or negative acknowledgement signal to be generated. This provides the static inputs necessary for the selection logic that is implemented using dynamic logic.

Two problems are likely to limit the performance of such an approach. Firstly, the architecture is complicated by the lack of a generalised register renaming scheme. This both limits the extent to which ILP may be exploited and complicates the process of determining when an instruction may be dispatched. Secondly, the resolution of dependencies using a counterflow pipeline is likely to represent a significant delay between the dispatching of some dependent instructions. This again will limit the scheduling hardware's ability to exploit ILP. From an implementation perspective the architecture requires a very wide counterflow pipeline, around 350 bits for an architecture with 64-registers, 32 window entries and 4 functional units. No results from any performance studies have been presented so far.

The provision of a data forwarding scheme is also not discussed. The data-forwarding scheme presented in the previous section relies on the availability of information indicating the source of each result. This type of information is lost if dependency resolution is performed using unary encoded vectors. Due to the relative implementation cost of dynamic scheduling hardware and data forwarding, it is probably sensible to consider an efficient data-forwarding scheme before devising a dynamic scheduling mechanism.

The scheme could be adapted to operate in the generic superscalar framework described previously in this chapter. The counterflow pipeline is retained, while unary encoded dependency vectors are replaced by result destination identifiers. Again the major concern would be the latency involved in communicating wake-up information to newer instructions. The counterflow pipeline experiments that have already been performed tend to suggest that its performance would not be adequate for this purpose. While the provision of a separate reorder buffer would allow full-compaction within the scheduling window, a simpler non-compacted implementation such as the one described previously in this chapter, could perhaps compensate for the lack of compaction with a slightly larger number of entries.

3.5.2 The FRED architecture

The FRED architecture [119, 118] is shown in Figure 3.17. A central dispatch unit together with a register scoreboard implements out-of-order instruction issue. Implementation details are scarce, and only a brief description of the problems of providing arbitrated access to the shared resources of the dispatch unit is given.

Again the lack of register renaming means that both true and false-dependencies must be tracked within the dispatch unit.

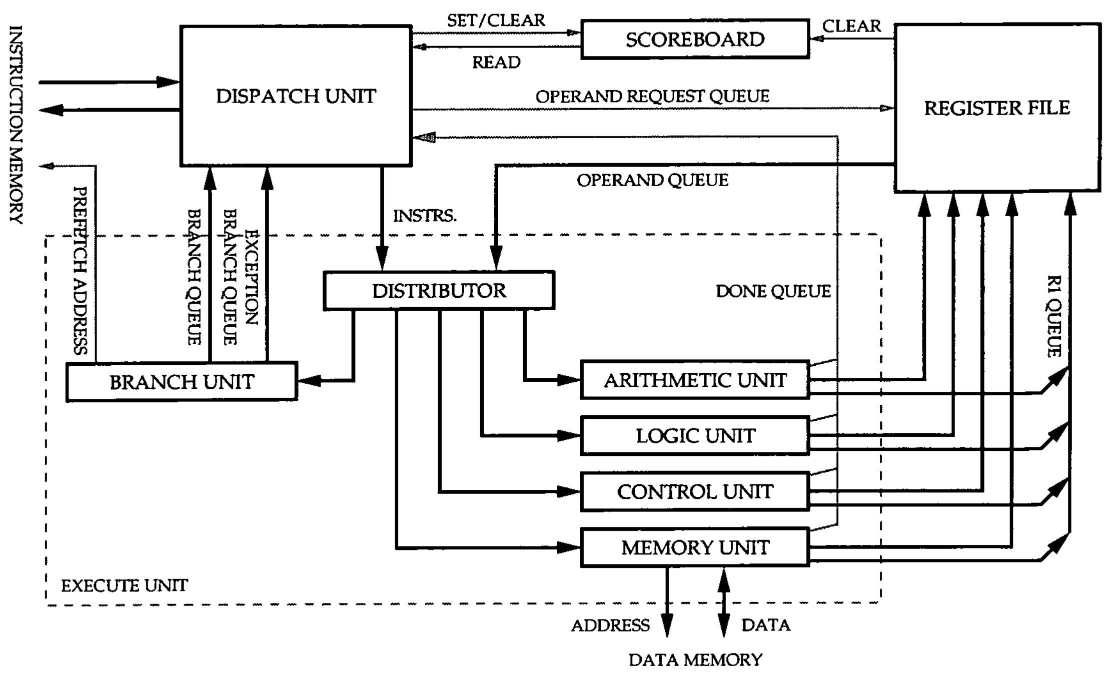


Figure 3.17: FRED Architecture

One unusual feature of the architecture is that register $R1$ accesses a queue. Writing to register $R1$ adds data to the tail of the queue, while specifying $R1$ as an operand will cause a value to be read out from the head of the queue. To ensure deterministic operation, instructions that utilise the queue are forced to execute in the original program order. Deadlocking the processor by filling the $R1$ queue must be avoided by the programmer, although such a condition would also force an exception. The $R1$ queue is available to every functional unit, as shown in the block diagram (queues are not represented explicitly as every data and control path is potentially buffered).

A very restricted form of data-forwarding is implemented. Instructions are able to reuse the last result generated at the functional unit they are dispatched to, although conditions are described when even this data cannot be used.

Again, the implementation of a dynamic scheduling mechanism is likely to be a poor replacement for a complete data-forwarding scheme. Some form of register renaming is also likely to boost ILP and simplify the implementation of the dynamic scheduling hardware. As a result of these problems, performance is reported to be lower when out-of-order issue is used.

3.6 Summary

In many cases the adoption of an asynchronous design style will require architectural modifications in order to fully exploit its potential advantages. In particular, the removal of high-level synchronisations may force the design to operate in a pseudo-synchronous fashion. In the case of a superscalar architecture, the existence of a large number of parallel instruction pipelines makes this task even more important. Two mechanisms key to exploiting ILP in such processors present a challenge in this respect. Both dynamic scheduling and data forwarding schemes traditionally exploit synchronous operation, their inclusion in an asynchronous ILP architecture without careful design will force the majority of datapath components to operate in lockstep – exposing both worst-case delays and control overheads.

Solutions to both problems have been presented in this chapter. Traditional implementations of dynamic scheduling exploit global synchronisation to control access to the dispatch buffer creating sequential wakeup and selection phases. In the asynchronous design each process that must access the buffer operates concurrently. Mutually-exclusive access is only enforced when it is necessary to guarantee a successful communication between two dependent instructions. A design is also presented for the selection logic that removes the need to implement a large N-way arbiter. In this case, synchronous operation is mimicked by sampling the state of the instructions before selection takes place.

In providing a data forwarding mechanism, two approaches may be employed to substitute for the lack of global synchronisation. Firstly, if the order of particular operations may be guaranteed we may be able to exploit this information even in an asynchronous environment. This leads to a simple data-forwarding mechanism if dispatch is guaranteed to be in-order at each functional unit. Secondly, information may be maintained locally to minimise the need for additional synchronisations in order to acquire non-local state information. A data forwarding scheme to support out-of-order dispatch is possible, if instructions are tagged with the source of data while they are stalled in the dispatch buffer. This data may then be used after dispatch to acquire operands via forwarding.

In the following chapter we will expand these ideas to develop an architecture that is targeted specifically at an asynchronous implementation. In particular, we focus on how the datapath may be simplified by exploiting explicit dependency information appended to instructions during compilation.

Chapter 4

Compounded Instruction Architectures

4.1 Introduction

The previous chapter explored how a typical synchronous architecture could be transformed to operate efficiently without reference to a global clock. Actual delays exposed by operating asynchronously are exploited by maximising concurrency – creating opportunities for local timing gains to be translated into increased overall performance. This is achieved by distributing both control and state to minimise the need for high-level synchronisations. Additional concurrency may also be exposed by reducing the need for mutually-exclusive operation. One way in which this may be achieved is by considering the need for mutual-exclusion as a data-dependent requirement.

The aim of this chapter is to explore how information appended to instructions at compile-time may also aid in exploiting asynchrony. The ability to append information prior to instruction fetch or issue can be viewed as an extension of the idea of distributing state and control. The previous chapter demonstrated how maintaining a small amount of state locally in the issue unit could provide a mechanism for implementing data-forwarding. One limitation of such an approach is that information can only easily flow in one direction: from older to younger instructions. In contrast, information appended at compile-time may exploit a global view of the program, albeit without precise information regarding the behaviour of branches and caches.

The goal of exploring such an approach is an attempt to find novel schemes for dynamic scheduling and data-forwarding specifically aimed at an asynchronous implementation. One target, in particular, is to simplify the implementation of both the wake-up and selection processes. In the scheme detailed in the previous chapter, these processes operate on all entries of the dispatch buffer in parallel. For this reason the implementation can be thought of as a brute force technique, utilising a significant level of power and offering poor scalability. In the synchronous case, there is little alternative as the operations are designed to operate within a fixed time period. Adopting an asynchronous design style allows additional performance/complexity trade-offs to be explored.

The technique that forms the basis for the architecture described in this chapter is called *instruction compounding.* In contrast to a synchronous VLIW architecture, where grouping independent instructions is beneficial, we aim to exploit explicit dependency information to simplify execution. This involves identifying groups of dependent instructions that are then scheduled as atomic units called *instruction compounds.* This additional dependency information may then be exploited at run-time by an asynchronous processor. A brief summary of some of key features of an instruction compounding processor is provided below.

- Dependent instructions are grouped at compile-time into instruction compounds.
- Data is only forwarded between consecutive instructions in compounds. Communication between compounds is performed through the register file.
- Explicit dependency information allows the producer of a result to make a request to forward its data to a consumer.
- Requests to forward data are also used to wake instructions up, indicating that one of their operands is ready. This mechanism becomes part of the dynamic scheduling scheme, which allows instructions in different compounds to execute out of order.

4.2 Instruction Compounds

Instruction compounds consist of a number of instructions where each instruction in the group is dependent on the previous. In the first instance, compounding is restricted to within basic blocks, and will assume that compounds are formed at compile-time. Compounding instructions from different basic blocks is complicated by the presence of control hazards, which prevents the need for communications to be identified until run-time. We will also assume that the most efficient way to name operands in such communications is by using registers.

Figure 4.1 illustrates two possible ways in which compounds may be selected from the instructions a to g. The only restriction placed on the compiler while forming compounds is that the resulting compound dependency graph is acyclic. If this is not the case, as illustrated by the grouping on the right hand-side, it becomes impossible to schedule the compounds atomically.

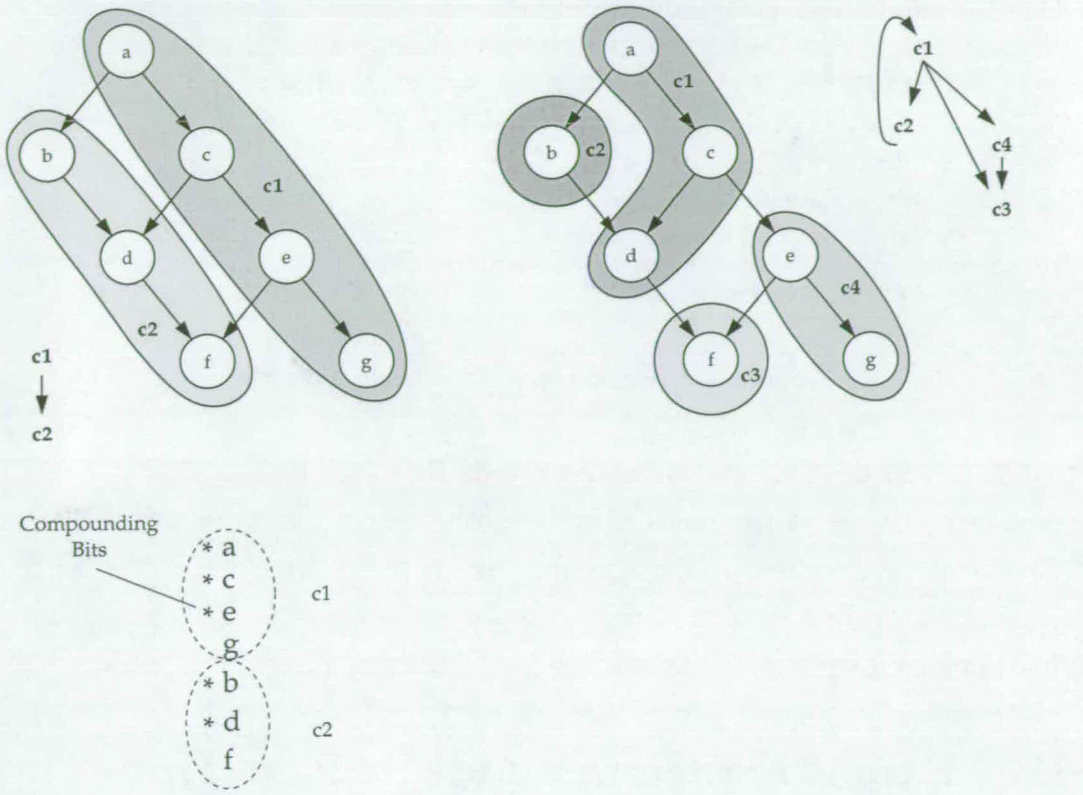


Figure 4.1: Two possible basic-block partitionings that form compounds. The compounds shown on the left form a valid cycle-free dependency graph, while the rightmost graph does not. The compounded program for the valid case is also listed.

Membership of a particular compound is indicated by a single additional bit appended to each instruction. This places the restriction that each member of a compound must be scheduled in order and consecutively. The setting of a particular instruction's compounding bit indicates that the next instruction is also a member of the same compound and requires the result of the preceding one. This form of compounding permits only a single communication from one particular instruction to be indicated explicitly. Figure 4.1 also shows the only valid schedule for the rightmost compound. In this example, instructions whose compounding bits that are set are preceded by an asterisk.

A simple graph partitioning algorithm that compounds instruction DAGs is provided in Figure 4.2. The $-\!\!\!\prec$ symbol is used to represent the covering relation, this evaluates true when there is a dependency between the two instructions, for example y depends on x ($x < y$), but no other instruction z exists such

```
COMPOUND($DAG, maxLength$)
$EdgeList$ = CREATEWEIGHTEDEDGELIST($DAG$)
SORT($EdgeList$)
/* Initially each instruction is a compound */
for each $(n_{from}, n_{to})$ in $EdgeList$
do Let $C_1$, $C_2$ be the compounds containing instructions $n_{from}$ and $n_{to}$
    if ((LENGTH($C_1$) + LENGTH($C_2$) $<=$ $maxLength$)
       and (ISTAIL($n_{from}$)) and ( not (ISSTORE($n_{from}$)))
       and (ISHEAD($n_{to}$)) and ( $C_1 \prec C_2$ ))
      then CONCAT($C_1, C_2$)
           DELETE($C_2$)

```

Figure 4.2: Basic block paritioning algorithm

that $x < z < y$. This prevents compounds from being created that cannot be scheduled. Edges of the DAG may be given weights to prioritise the creation of particular compounds. For example, to increase the chance of a particular communication being serviced using the forwarding mechanism, or for implying a particular set of dynamic scheduling possibilities.

4.3 Exploiting Compounds at Run-Time

In this section, we examine how the availability of explicit dependency information may be exploited at run-time. The value of such information is that it provides a means by which the consumer of a result may be identified without the need to introduce high-level synchronisations.

A simple technique to exploit such information, which would combine both dynamic scheduling and data forwarding, is to consider moving entire compounds between execution pipelines. The instruction issue stage would operate by initially sending the whole compound to the functional unit required by the head (first) instruction. Once the compound had reached the end of the dispatch queue and the head instruction had fetched its operands, its tail (i.e. all instructions except the first) would be forwarded to the functional unit of the new head instruction. As soon as the result of the first instruction had been generated it would be forwarded to the tail's functional unit. A number of architectures that operated in this manner were explored [7].

These architectures unfortunately suffer from two problems. Firstly, deadlock conditions could potentially arise when compound tails are sent between functional units. Compiler-based scheduling constraints must be introduced to avoid the possibility of deadlock or dynamic run-time schemes should be devised to detect deadlock situations. Scheduling constraints were explored, although it was concluded that opportunities to create compounds were limited if this approach was taken. The lack of an appropriate formalism and related tools also makes reasoning about the interaction of programs and such architectures difficult. Related work, described in [16], investigates the application of typing calculi to prove correctness given a particular architecture and program. A second problem is the impact on power and performance of moving a large number of instructions between functional units. In this respect, a design that issues instructions once to the functional unit at which they are required would be preferable.

4.3.1 Overview of a compounding architecture

Figure 4.3 provides an outline of an architecture that is designed to exploit instruction compounding. In some ways this architecture appears similar to the queue-based architecture described in Section 3.4.1. While there are similarities, the compounding architecture is capable of dispatching instructions out-of-order from each dispatch buffer. Data-forwarding is also implemented using a completely different mechanism. This section provides a brief overview of the different components of the architecture.

Not shown on the diagram are the additional components required to support speculative execution and register renaming, which include the register rename mapping tables, branch prediction logic, free register pool and reorder buffer. We assume that their implementation poses few problems, and they are therefore omitted for the sake of clarity.

Instructions are fetched and proceed in parallel to the register rename stage. At this point, logical registers are mapped to physical registers and the busy-bits corresponding to each instruction's destination register are reset. Note that unlike the dispatch buffer described in the previous chapter, the busy-bit table performs no arbitration and is simply implemented as a multi-ported memory. After register renaming the instructions proceed in parallel to the instruction issue unit.

Instruction issue consists of two stages. Firstly, if the instruction is a member of a compound, then the location of the next instruction (consumer) in the compound is appended to it. The location takes the form of both a functional unit identifier and a dispatch buffer entry index. The dispatch buffers themselves are organised as parallel FIFOs that allow the next free entry in each buffer to be determined using counters maintained in the issue module. The correct buffer index for the consumer instruction may therefore be obtained as soon as the functional unit it requires is known. If both instructions in question were fetched in parallel and are now both in the issue stage then this operation is trivial. If the compound spans an instruction fetch boundary, the type of the next instruction must be obtained from the register rename stage. The second step in the issue process simply distributes each instruction to the appropriate functional unit. If functional units are duplicated, then we assume instructions are issued in a round-robin fashion.

A compounded instruction encapsulates information about the use of results of instructions within the compound; this enables data forwarding operations to be initiated by the producer of results. In contrast, in all the architectures discussed previously in Chapter 3, it is the consumers of results which request data from the producer (or producer's FU). The basic steps of a forwarding operation are illustrated in Figure 4.3. The highlighted control and data paths around $FU2$ represent the communications necessary to implement a forwarding operation. On dispatch, a compounded instruction will initiate both its operand fetch stage and also its *forwarding request unit.* The instruction is then able to make a request to forward data to the next instruction in the compound. The request is directed at the correct dispatch buffer entry using the location information appended during issue. A response generated at the consumer's dispatch buffer then determines if forwarding proceeds or not. The forwarding request now also forms part of the wake-up process of the consumer instruction, indicating that one of its operands is ready. The following sections provide a detailed description of how these dynamic scheduling and data-forwarding processes operate and may be implemented.

4.3.2 Out-of-order dispatch

A dynamic scheduling mechanism exposes ILP by enabling instructions from many compounds to execute in parallel. One aim in developing such a scheme is to avoid the need to broadcast result register identifiers to all the waiting instructions and the resulting high fan-in associated with the selection logic. The scheme must also be compatible with the way in which forwarding requests now emanate from the instructions producing the results. This is achieved by updating the status of operands of individual instructions directly, without the aid of comparators or content-addressable memory. The wake-up operation is now little more than a write to a single memory location in one of the dispatch buffers. In the following discussion, the term *micro-operations* defines the constituent steps for executing an instruction. A micro-operation may travel though a number of

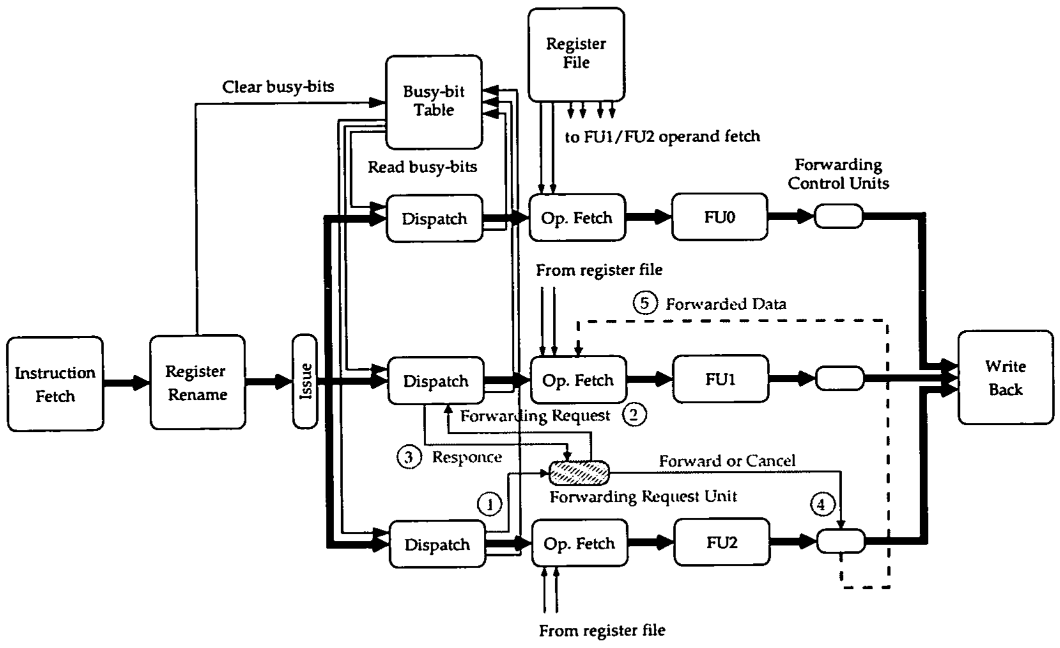


Figure 4.3: A superscalar instruction compounding processor. For clarity the data buses required for data forwarding have been omitted.

local control units to complete its task or may simply need to complete a single handshake with one particular function unit.

Wake-up operations may be generated either by micro-operations which have read a ready entry in the Busy-bit Table, or by forwarding requests. The micro-operations for reading the Busy-bit Table are generated and queued for each operand in the buffer that will be obtained via the register bank. These represent all communications that are not explicitly indicated through compounding. Each read micro-operation accesses the busy-bit table and waits until the entry corresponding to the particular operand's register is set. This indicates that the instruction that will generate the result for the register has been dispatched. Once the busy-bit has been read as set, then the wake-up operation may proceed. The second way in which operand status is updated is through forwarding requests. These emanate from the preceding instruction in an instruction compound. The precise handling of forwarding requests is described in detail in the next section.

A block diagram of the logic surrounding a single dispatch buffer is shown in Figure 4.4. Two busy-bit read queues are used to store the pending read and wakeup micro-operations. Each micro-operation contains the operand's register identifier and its entry in the buffer. Operations are allocated to the queues in round-robin style. No arbitration is required at the busy-bit table as read operations simply stall until the bit they are reading is set, which is identical to the behaviour of the lockable register file. The other source of wake-up operations is provided by forwarding requests. Forwarding requests from each of the functional units are first arbitrated and then buffered before being served in sequence by the dispatch buffer. In this case the single `ack` signal is replaced by both `ack` and `cancel` signals as in some cases forwarding is not allowed to proceed as detailed in the next section.

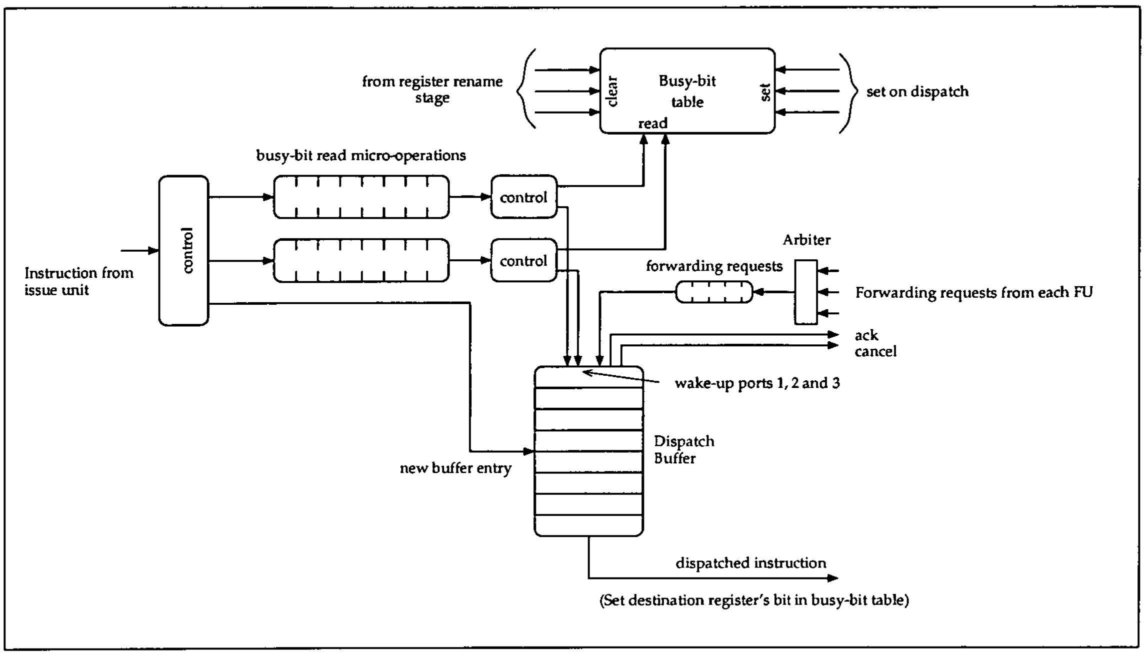


Figure 4.4: Reading busy-bits and waking buffer entries

Figure 4.5 provides an example of how an instruction may be dispatched in the compounding processor. Operations which may be performed in parallel are composed using the parallel operator (||), while those which must be performed sequentially are composed using a sequential operator (;). In the example, communication is performed through the register-file as the instructions do not belong to a compound. Information is communicated between the instructions at two points. Firstly, the dispatch of the first instruction is detected by the second instruction after its busy-bit is read as set. Secondly, the result of the first instruction is read by the second during its operand fetch stage. A register-locking mechanism, as described previously in section 2.6.2, ensures that the second instruction only reads the data after the first has finished its write-back operation.

```
A = ADD R5, R6, R7
B = LD R1, [R5]

// Initially
Busy bit 6,7 is set
Busy bit 5 is clear
Register R1 and R5's lock bits are set on issue

// Execute A
(
((write instruction A to dispatch buffer entry N, initialise
  each operand's status to not ready)||
 (queue micro-operation to read busy-bit 6, for entry N)||
 (queue micro-operation to read busy-bit 7, for entry N));
(((wait until busy-bit 6 is read as set);
  (update status of op R6 in A's reservation station)) ||
 ((wait until busy-bit 7 is read as set);
  (update status of op R7 in A's reservation station)));
(dispatch A);
((set busy bit 5)||(read operand R6)||(read operand R7));
(execute A);
(write back result to R5, clear R5's lock bit)
)

// Execute B
(
((write instruction B to dispatch buffer entry P, initialise
  each operand's status to not ready)||
 (queue micro-operation to read busy-bit 5, for entry P));
(wait until busy bit 5 is read as set);
(update status of op R5 in B's reservation station);
(dispatch B);
// Note: operand fetch may be stalled until valid data
// is available, this is enforced by the setting of lock
// bits in the register file
((set busy bit 1)||(read operand R5));
(execute B);
(write back result to R1, clear R1's lock bit)
)
```

Figure 4.5: Example of instruction dispatch in a compounded instruction architecture. In this example all the communications take place via the register file.

The wake-up logic for a single reservation station is shown in Figure 4.6 (the logic for a broadcast style reservation station is shown in Figure 2.29). Each of the wake-up operations, produced either as a result of busy-bit reads or forwarding requests, access a particular wake-up port on the buffer. There are three identical wake-up ports associated with the two parallel busy-bit read operations and the incoming forwarding requests. Wake-up operations when performed at the buffer, update the status of the target operand directly. This creates a request (rising edge) at the input to one of the OR gates shown in the diagram, setting the associated status flip-flop. When both of these status bits have been set the instruction is ready to be dispatched.

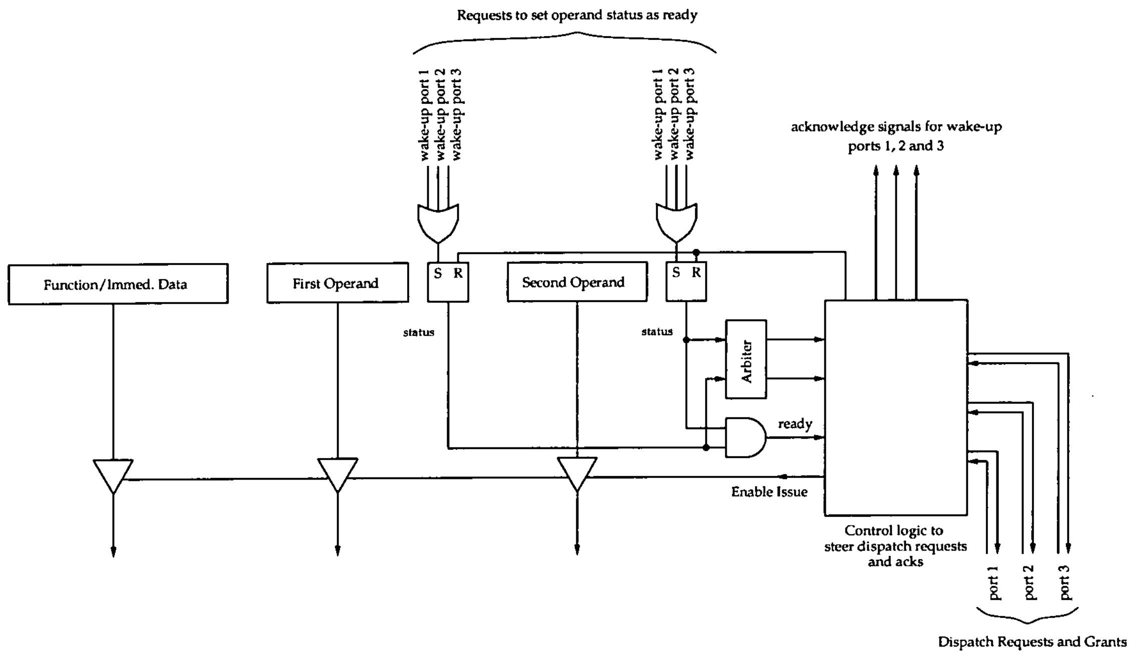


Figure 4.6: Wake-up logic for a single reservation station

If wake-up operations were acknowledged immediately by the buffer, then the selection logic described in section 3.3.2 would be required to select a ready instruction for dispatch. In practice, the selection logic is simplified by limiting the number of instructions that may become ready simultaneously. To ensure this, the last wake-up operation to update the status of an instruction's operands is not acknowledged until the instruction has been dispatched. This prevents the wake-up port from being used to wake-up another instruction. The number of simultaneous dispatch requests is therefore limited to the number of wake-up ports. The arbiter shown in the diagram is required to determine which operation was the last to update the status of the instruction, as wake-up requests to the same instruction may be made concurrently at different wake-up ports.

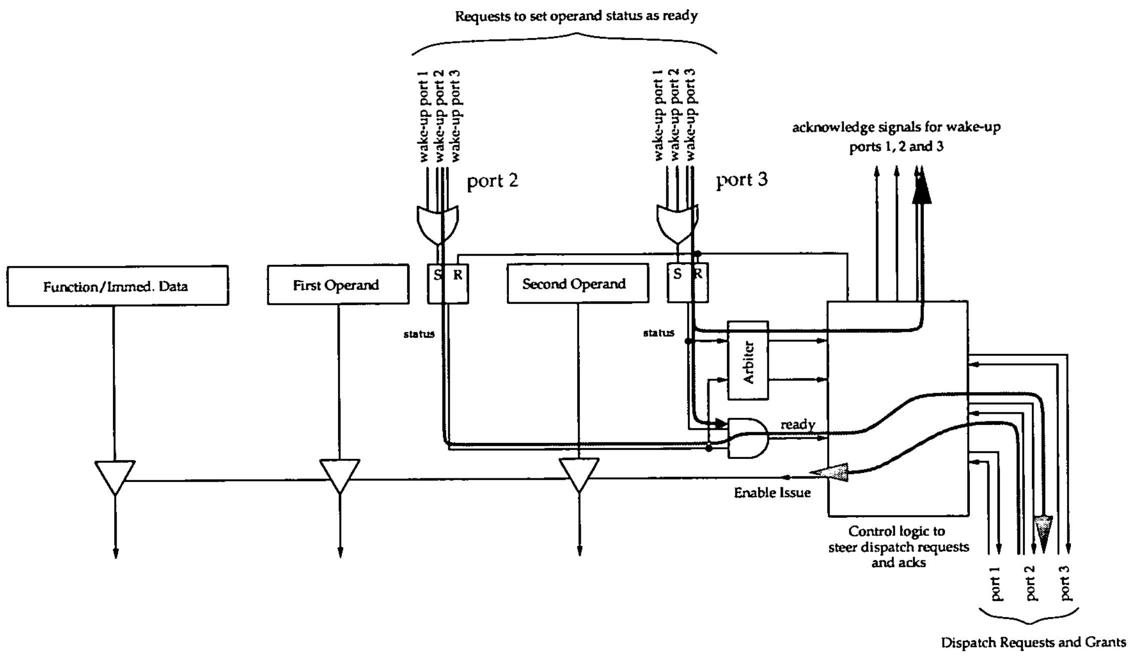


Figure 4.7: Wake-up requests and acknowledgements

The wake-up process is illustrated in Figure 4.7. Two concurrent wake-up requests are made to the reservation station; in this case, through wake-up ports 2 and 3. Once the status of both the instruction's operands have been updated, then the instruction is ready for dispatch. The arbiter allows one of the wake-up ports to be acknowledged (in this case port 3), allowing it to be used again immediately to wake-up further instructions. The other port is stalled until the instruction in the reservation station has been dispatched. A dispatch request is initiated on the dispatch port corresponding to the stalled wake-up port – port 2, in this case. The instruction is dispatched as soon as a grant is received, the wake-up port is then acknowledged.

A block diagram of a complete dispatch buffer is shown in Figure 4.8. In this example, each reservation station is capable of raising one of three dispatch requests. These outputs are OR'ed together and arbitrated in order to generate each dispatch grant signal. At most one dispatch request will be granted, allowing at most one instruction to be dispatched at a time.

4.3.3 Data forwarding

The architectures described in Chapter 3 queue forwarded results at their destination depending on their source. These queues are required as the order in which results are received from different sources cannot be guaranteed to be the same order in which they must be consumed. In these architectures this is the only

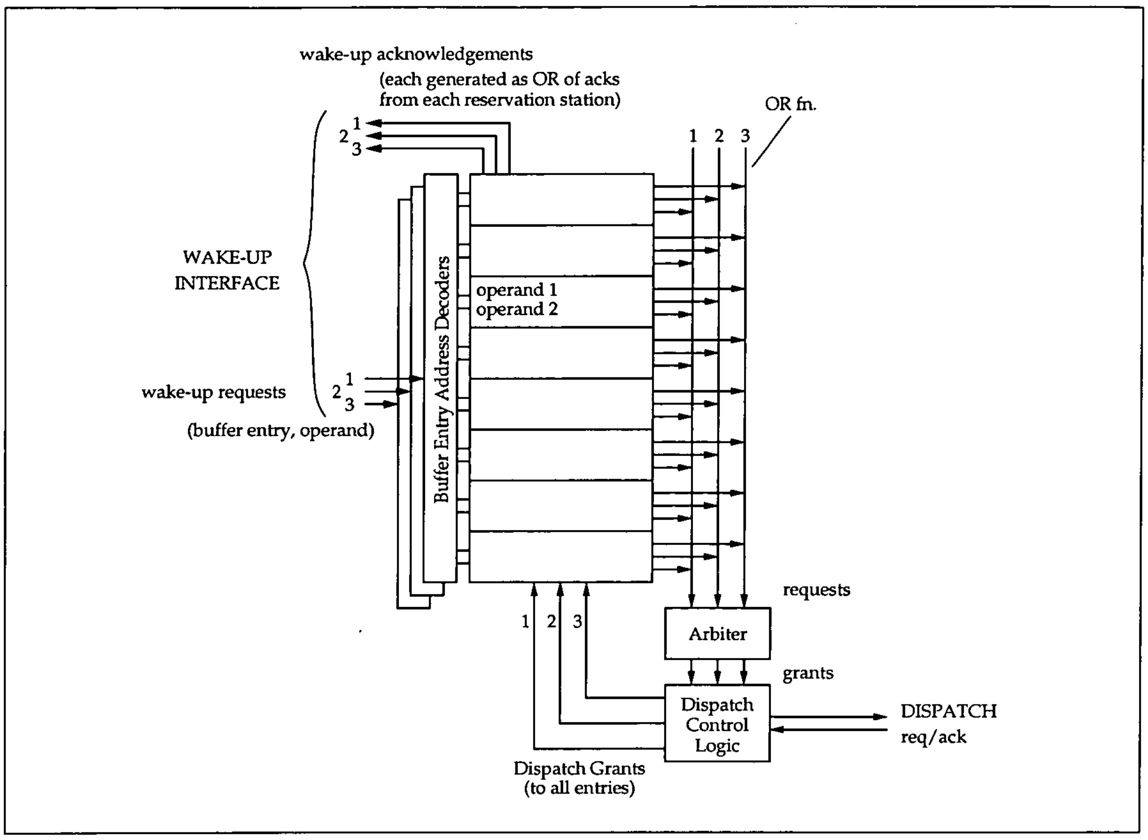


Figure 4.8: Dispatch Buffer

step which must be taken to ensure that instructions receive the correct data. This is because the order in which instructions arrive at their operand fetch stage can be controlled, either because dispatch is in order at each functional unit, or because forwarding requests are made after instructions are dispatched (by those instructions requiring the results).

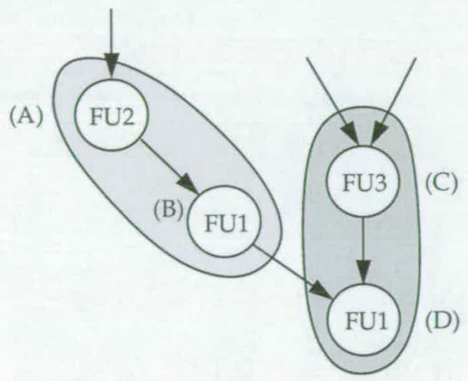
In the compounded instruction architecture additional care must be taken to ensure that forwarded data reaches the correct instruction. Only when a forwarding request causes the instruction to make a dispatch request can we guarantee that the instruction will receive the correct forwarded data. If a forwarding request is made to an instruction which is not ready to be dispatched, we must create a forward cancel signal in place of the acknowledge signal. In either case the status of the corresponding operand is updated to indicate that it is ready. If the forwarding operation has to be cancelled, the operand is obtained via the register bank. If forwarding can proceed, the result is read from the appropriate forwarded data result queue. To ensure that only valid data is read from the register file, each register is locked using a single bit as described previously.

Another plausible option may be to stall the wake-up port associated with the forwarding request and wait until the instruction becomes ready. This option is undesirable as it may lead to deadlock, this is a possibility as the dispatch buffer is blocked from accepting additional forwarding requests. Blocking such requests may mean execution can no longer proceed. Such an approach, even if it did not lead to deadlock, would probably lower performance by attempting to forward data to instructions that may not be dispatched until the data is available from the register file. An example of how compounds execute on such an architecture is provided in the following section.

4.3.4 An example

This section provides a step-by-step description of how instruction compounds are executed. The program fragment, together with compounding bits and target FU, used to illustrate the dynamic scheduling and forwarding mechanisms is shown in Figure 4.9. Also shown is a sample compound selection and a description of the format used to represent the state of the busy-bit table read queues and dispatch buffer.

	C-Bit	Instruction	FU
(A)	1	MUL R1, R2, 7	2
(B)	0	LD R3, [R1]	1
(C)	1	ADD R4, R5, R6	3
(D)	0	ST R4, [R3]	1



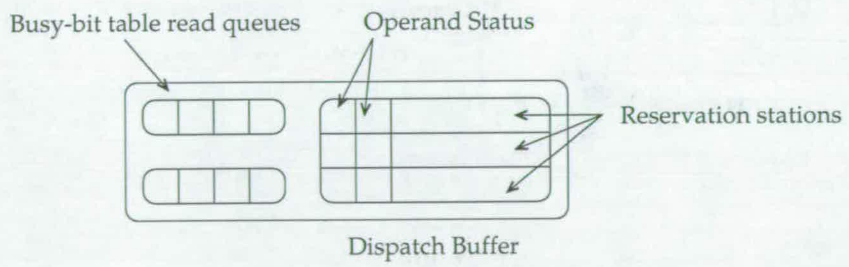


Figure 4.9: Example program fragment and explanation of format used to represent state at each functional unit

Figure 4.10 Step 1, shows the initial state of the busy-bit table read queues and dispatch buffers after all the instructions in the program fragment have been issued. Busy-bit read operations are generated and queued for each of the register operands in the fragment. Each operation contains the register identifier and buffer entry of the corresponding instruction. Communications that are initially

set to take place via the data forwarding mechanism do not generate entries for the read queues.

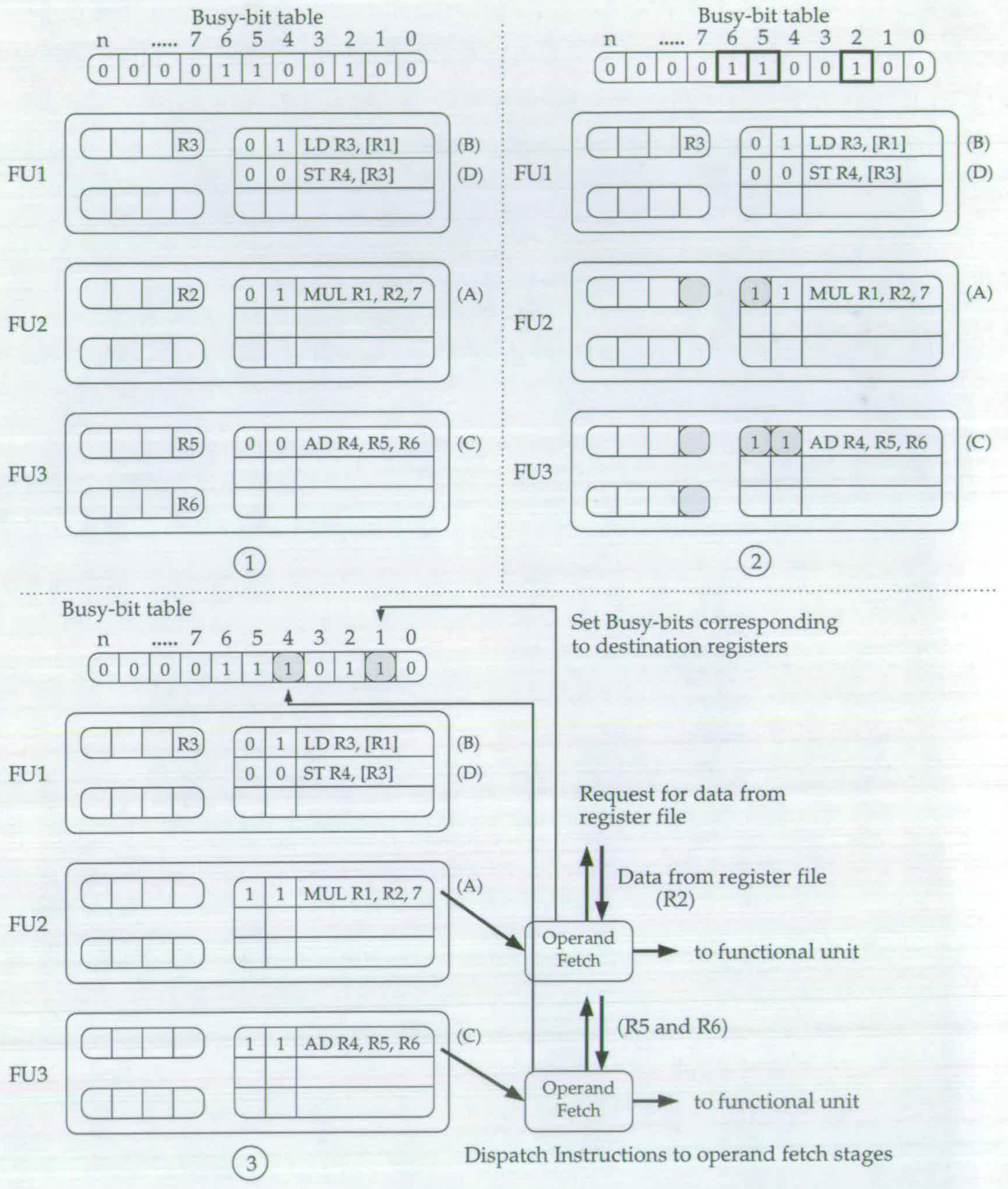


Figure 4.10: Sample execution of a compounded instruction

The instructions A and C are tagged with the location (FU and buffer entry; the tags are not shown in the diagram) of the next instruction in their compounds, in this case B and D respectively, this information is stored together with the instruction in its dispatch buffer. The status of all the register operands in the dispatch buffer is initially set to zero, indicating that the instructions are not yet ready to be dispatched. The busy-bit table indicates that the registers required to begin execution of instructions A and C are ready ($R2$, $R5$ and $R6$).

Step 2, shows the state of the datapath after all the busy-bit read and wake-up operations for functional units 2 and 3 have completed. This updates the status of the operands of both instructions A and C making them both ready to be dispatched. The shaded areas in the figure are used to highlight these state changes. The busy-bits read are also highlighted with thick black borders.

In Step 3, both instructions A and C are dispatched. At this point each instruction sets the busy-bit corresponding to its destination register. Requests are also made to the register file for each instruction's operands, in this case registers $R2$, $R5$ and $R6$.

In parallel with the operand fetch stage outlined in the previous step, both instructions A and C initiate forwarding requests. The order in which these forwarding requests are generated depends on when each instruction is actually dispatched, the queuing of such requests requires arbitration. In the first instance we consider the scenario when the forwarding request from functional unit 2 (instruction A) arrives first and is placed at the head of the forwarding request queue at Functional Unit 1 (as illustrated in Step 4 of Figure 4.11).

The forwarding request is tagged with the location in the dispatch buffer of the instruction requiring the result. When the request reaches the head of the forwarding request queue a wake-up operation is performed. The status of instruction B's operand $R1$ is updated to indicate that it is (or soon will be) available. In this case as the data is to be acquired through data forwarding the source of the data (functional unit identifier) is also appended to the instruction.

Updating the status of $R1$ allows instruction B to request to be dispatched. As the forwarding request renders the instruction ready to execute, it is acknowledged for the data must now be sent. The final step illustrates how instruction B obtains its operand via the data forwarding mechanism. Instruction A first sends its result to the forwarded result queue corresponding to Functional Unit 2; this data is then read when instruction B reaches its operand fetch stage – the correct queue is selected using the information received during the forwarding request. The final instruction D will dispatch soon after B has set the busy-bit corresponding to its destination register and its forwarding request has been serviced. Instruction D will also read forwarded data, in this case from the forwarded result queue corresponding to Functional Unit 3.

Another possible scenario is that the forwarding requests illustrated in step 4 arrive in reverse order, resulting in the forwarding request from instruction C being queued ahead of the request from instruction A. Figure 4.12 illustrates how deadlock is avoided in this case. Instruction D cannot be permitted to be dispatched in this case as instruction B must execute first (to generate the result for register $R3$). As D cannot be dispatched, the forwarding request must be cancelled. The wake-up operation is still permitted to change the state of operand $R4$, although the data will be obtained through the register file when instruction D is eventually dispatched.

Once the forwarding request for instruction D has been handled, the forwarding request for instruction B proceeds and is successful (illustrated in step 7). As soon as B is dispatched, the busy-bit for register $R3$ is set that will eventually allow instruction D to make a request to be dispatched. When D reaches its operand fetch stage both operands will be read from the register file as the forwarding operation has been cancelled.

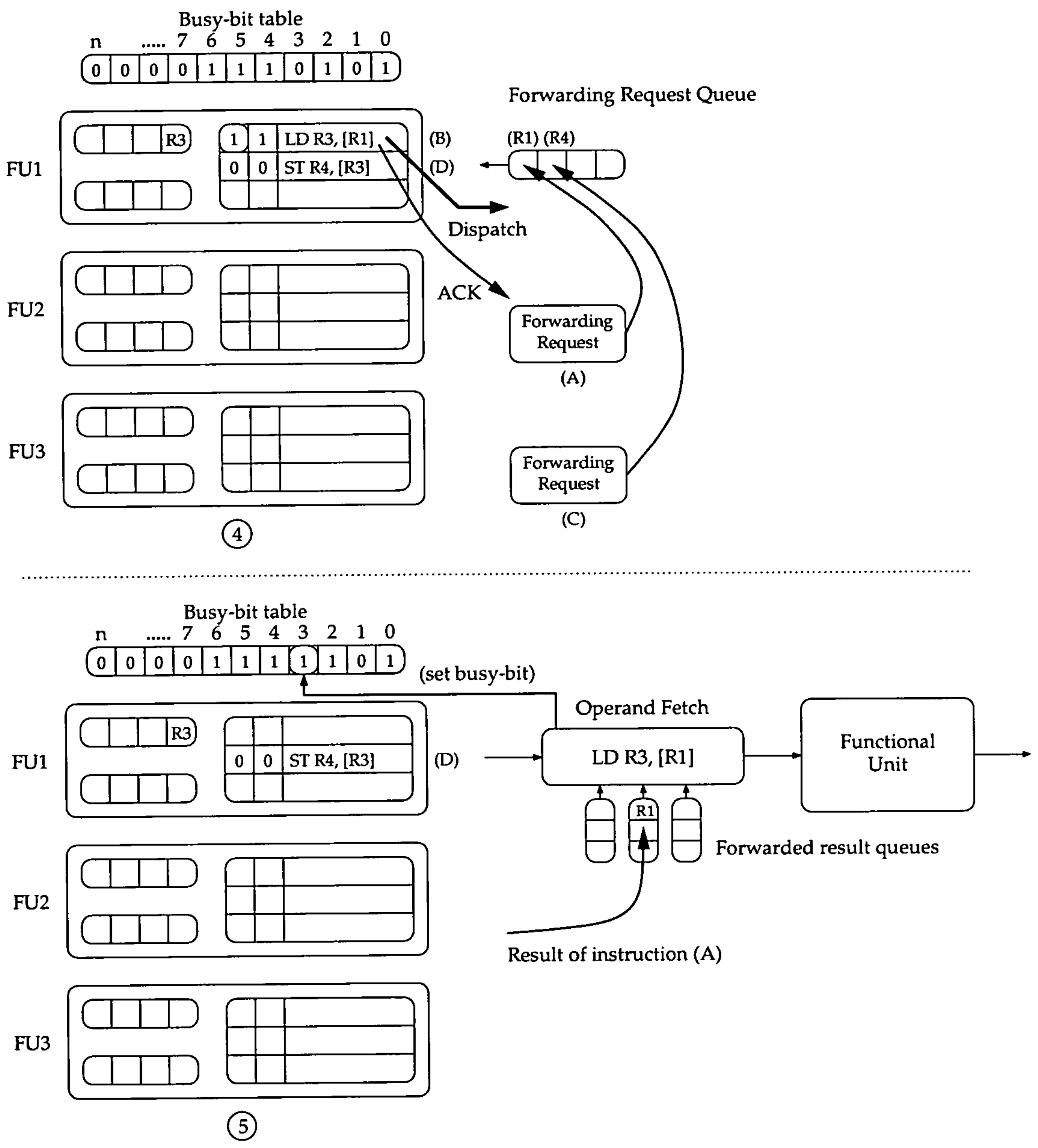


Figure 4.11: Sample execution of a compounded instruction (continued)

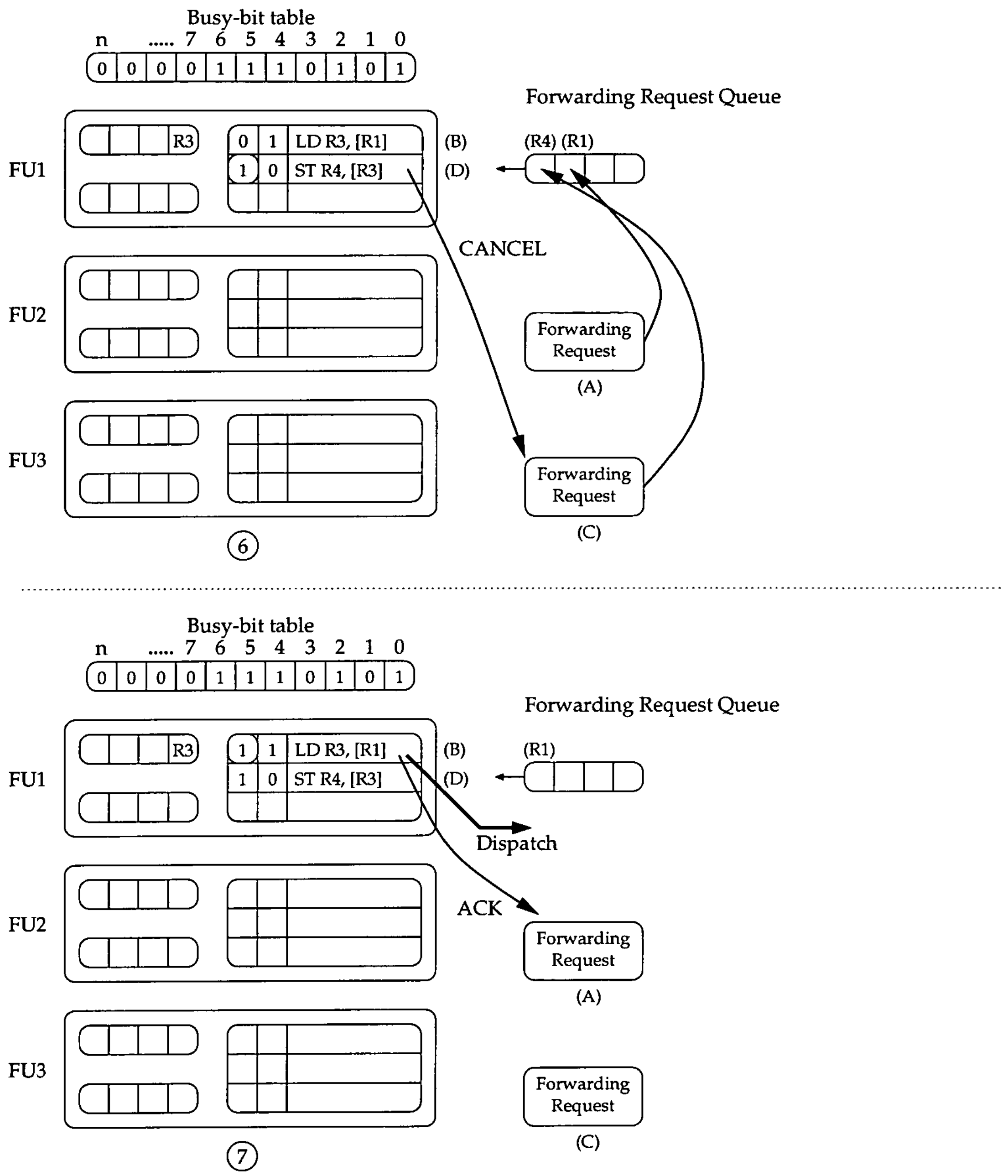


Figure 4.12: Sample execution of a compounded instruction (cancelling a forwarding operation)

4.3.5 Load and Store Operations

Section 2.8.5 described a number of techniques that are used to schedule memory operations. As in synchronous processors, the dispatching of load instructions in a compounding datapath is preferably handled in a speculative manner. Correct program semantics is ensured by detecting store-to-load dependencies that have not been respected after a load has been dispatched. Recovery mechanisms such as those described in chapter 2 are then used to rollback the execution to a point before the load was dispatched. Schemes that attempt to respect all memory dependencies by comparing addresses prior to dispatch would provide significant implementation problems, while also unnecessarily stalling the dispatch of the majority of load instructions unnecessarily. The frequency at which loads and stores that are executed in proximity, reference the same memory location, should in practice be small as such communication does in most cases take place through the register-file. Of course, situations do exist where it is impossible for the compiler to disambiguate memory addresses statically. The following section gives an overview of the implementation of the memory unit and describes how various techniques commonly found in a synchronous implementation may also be applied in an asynchronous one.

The execution pipeline of the memory unit is typically longer than that of a typical integer functional unit. In the compounded instruction architecture, the operand fetch is followed by memory address calculation and two cache access pipeline stages. In addition, address comparisons are required to ensure that all memory carried dependencies (store-to-load dependencies) are respected when loads are dispatched out of order. One possible scheme for performing such address comparisons is described below.

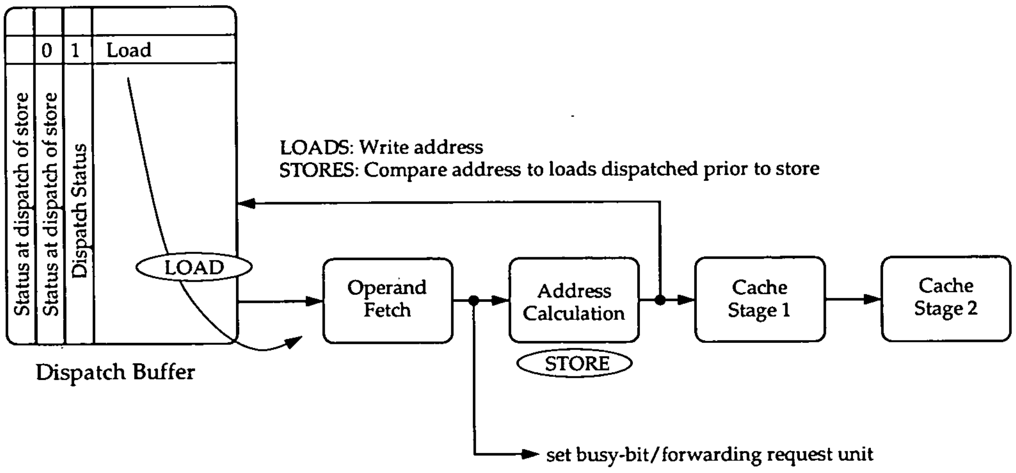

0
1
Load
Status at dispatch of store
Status at dispatch of store
Dispatch Status
LOAD
Dispatch Buffer
LOADS: Write address
STORES: Compare address to loads dispatched prior to store
Operand Fetch
Address Calculation
STORE
Cache Stage 1
Cache Stage 2
set busy-bit/forwarding request unit

Figure 4.13: Proposed scheme for detecting store-to-load dependencies

Figure 4.13 shows the memory dispatch buffer followed by its execution pipeline stages. A check to ensure that no store-to-load dependencies have been violated is made each time a store instruction calculates its effective address. At this point the address is compared to all younger loads that have been dispatched prior to the store instruction. Of course, store instructions can only execute in program order after their non-speculative status has been confirmed. To implement such an in-order mechanism, the stores are only dispatched after they receive a wake-up signal from the reorder buffer. This store wake-up signal is provided with its own dedicated wake-up port at the memory unit. The current dispatch status of loads is copied just prior to the dispatch of a store. The store then uses this copy of the dispatch status when it subsequently requests that its address be compared with younger dispatched loads. This ensures that loads dispatched after the store, but before the address comparison request is made, are not included in the address comparison. To complete the scheme, loads must write their addresses back into the dispatch buffer after they are generated.

4.3.5.1 Memory unit optimisations

In this section we discuss how schemes typically employed to improve memory performance in a synchronous superscalar processor may be adapted to operate in an asynchronous compounded instruction one. The three techniques are: hit/miss cache prediction, the use of a load wait table and aborting instructions on a cache miss.

The ability to predict a cache miss allows the dispatch of a load instruction's children to be delayed, and potentially allows other independent instructions to proceed. The scheme operates by preventing the load instruction from updating its corresponding busy-bit, or making a forwarding request, until after the prediction has been read. In the event that a miss is predicted the updating of the busy-bit, or forwarding request, is delayed until just prior to the result becoming available. The cache hit/miss prediction table is read after the address calculation and is updated after each memory operation. A simple predictor mechanism may be implemented by maintaining a counter for each recent load operation. The scheme utilised in the Alpha 21264 [75] uses a 4-bit saturating counter. Here the most significant bit of the counter is used as the load hit/miss prediction. The counter decrements by two on each cache miss and is incremented on a cache hit.

A second technique that may be used to improve the performance of the memory unit is the introduction of a load wait table [75]. This attempts to minimise the number of loads that are dispatched prematurely ahead of older stores. This table is read during the instruction fetch or register rename stage and is used to set a single bit in each load instruction. If the load has previously caused an exception then its dispatch is delayed until all previous stores have been executed.

One simple scheme to ensure that all previous stores have executed is to stall the dispatch of the problematic load until it reaches the head of the dispatch buffer. The dispatch port used to dispatch stores in order may be used for this purpose.

A more elaborate scheme is required if we wish to dispatch the load earlier, as soon as all previous stores have been dispatched. The detection of the condition when no stores exist between a particular delayed load and the head of the FIFO requires the generation of a *no store* signal. This signal is propagated along the FIFO from the head, to indicate that no store instructions exist between the current entry and the FIFO head. Entries in the buffer that contain a load that does not have an entry in the wait table, propagate the signal when it is received. In the other cases, where the entry is a store or a load that must be delayed, the signal propagates no further. Only when a load that is delayed detects that the *no store* signal is set can it make a dispatch request. Again an additional dispatch port would be required for loads dispatched in this manner.

The final technique aims to prevent cache misses stalling other functional units' execution pipelines. If a cache miss is detected, it may be beneficial to abort the execution of instructions that have been dispatched in anticipation of the load quickly producing a result. In a compounded instruction architecture this requires two problems to be solved. Firstly, how the consumer of the load's result is aborted and secondly, how the instruction is then dispatched a second time when the result of the load becomes available. Further problems arise when one considers that the dispatch of the consumer of the load may result in further dependent instructions being dispatched.

Aborting instructions in such a situation may be feasible by propagating abort signals through the forwarding paths and the register file. On detecting a cache miss the load would write or forward a value indicating a miss, instructions receiving such a signal would then do the same - forcing all instructions that are dependent on the initial load to be aborted. Such a technique may involve significant additional hardware. An alternative may be to stall the setting of busy-bits, or the sending of forwarding requests when instructions are dependent on the result of a load. In practice, this may simplify the abort operation but lower overall performance.

Unfortunately, even if instructions could be aborted in a satisfactory manner, the dispatching of the instructions a second time would be equally complex. In the synchronous case these operations are far simpler due to the presence of global synchronisation and the ability to broadcast result register identifiers to wake-up instructions. Overall, it may be concluded that this final optimisation is probably not suitable for inclusion in a compounded instruction architecture. To compensate for any loss in performance, greater effort could be focused on the need for a hit/miss prediction mechanism.

4.4 Dynamic Compounding

In the previous sections it has been assumed that compounds have been selected from instructions within the same basic block. Therefore, no communication between basic blocks can be serviced using the data-forwarding mechanism. In this section we describe a scheme to allow the limits of compounding, if it were to be permitted across basic block boundaries, to be explored.

Dynamic compounding is achieved by maintaining a forwarding table within the processor. The table records pending results and a list of potential consumers. This information is then used to construct compounds dynamically free from the restrictions imposed on the compiler. The table is accessed twice by each instruction: once to indicate that it has generated a result and again prior to the generation of a forwarding request to obtain the location of a consumer. At this point, the entry would also be reset to indicate that the result is no longer available via the forwarding mechanism. An instruction that requires a result that the table indicates will soon be available, simply records its destination in the same table entry and sets its compounding bit. The necessary forwarding request would then be generated by the producer after reading the location of its recorded consumer from the forwarding table. As in the static case, dynamic compounding is restricted to only recording a single consumer of each result.

The performance cost of such a scheme would be the additional delay, incurred by accessing the forwarding table, prior to generating a forwarding request. Consumers would write their dispatch location into the table during instruction issue, while entries would be read and reset by producer instructions following their dispatch. Synchronisation between the issue unit and each functional unit would be minimised by arbitrating access to the forwarding table on a per-entry basis, this is in fact very similar to the organisation of the busy-bits described in Section 3.3.

A different approach would be to form compounds within a trace cache [121], which could be used to avoid the need to centralise the generation of compounds as described previously. Compounding opportunities identified within instruction traces could be indicated by appending the offset of the consumer to the producer instruction. During instruction issue these offsets would then be used to determine the dispatch buffer entry the consumer instructions would subsequently occupy. Compounding in this way would also remove the selection restriction that requires compounds to form an acyclic graph.

In the simulation results that follow we simply use the dynamic compounding mechanism to explore the limits of compounding. For this reason, when the forwarding table is accessed within the compounded instruction processor model no additional delays are imposed.

4.5 Summary

The instruction compounding technique described in this chapter allows a novel approach to be taken to both data forwarding and dynamic scheduling. Compounding dependent instructions allows the location of an instruction that makes use of a result to be appended to the producer instruction. This allows instructions within a compound to communicate using a data-forwarding mechanism. Furthermore, as forwarding requests are made directly to waiting instructions they may also be used as wake-up operations.

Dynamic scheduling is simplified by removing both the need for a large fan-out associative wake-up operation and the traditional large fan-in of the selection process. These simplifications should reduce both the delays associated with dynamic scheduling and its power requirements.

Chapter 5

Results

5.1 Introduction

The previous chapters described asynchronous architectural techniques aimed at exploiting ILP. The aim of this chapter is to provide quantitative measures of goodness for the different approaches through simulation. In particular, the impact of key architectural parameters are investigated, such as the number of reservation stations and the data-forwarding configuration, on the architecture's ability to expose and exploit ILP. The asynchronous processor's ability to exploit reduced functional-unit latencies or fewer register read ports is also explored.

5.2 Experimental Setup

The simulation environment for the experiments is outlined in Figure 5.1. The simulator used to generate the final timing information is driven by a trace of instruction and data addresses. Trace-driven simulations such as this generally simplify the modeling process and improve simulation performance. Any further studies of the architectures would most likely be oriented towards a final implementation. At this point performance studies would require more detailed data-dependent delay information, which is only available through the use of an execution-driven model.

The benchmarks are compiled using the GNU C compiler. The binary is then modified to generate instruction and data traces using the Wisconsin Quick Profiling Tool (QPT2). The QPT tool operates by inserting code into each basic block to record execution frequency and data memory addresses. The QPT program is part of a larger set of tools called the Wisconsin Architectural Toolset (WARTS) [36]. Actual traces are then generated by running the modified benchmark using the appropriate data set.

The simulator is fed with instruction and data traces, together with a disassembled version of the original binary. This allows the actual instructions that were executed in each basic block to be identified. In the case where an instruction compounding processor is being modeled, a compounded version of each basic block is fed to the simulator in place of the original instruction schedule. Data cache behaviour is modeled using the Dinero (IV) cache simulator [35]. This provides a highly configurable model of cache behaviour. Access to the simulator is provided through a number of simple function calls integrated into the main simulator. Instructions are simulated at a rate of around 1000 per second for the most complex of the asynchronous processor models using a 300Mhz UltraSPARC II system.

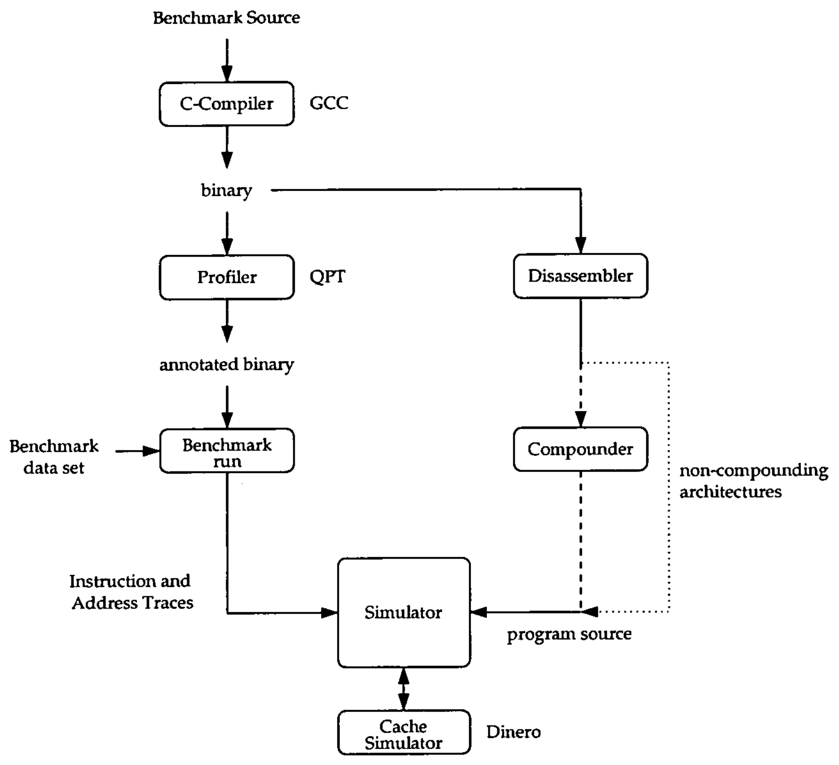


Figure 5.1: Simulation Environment

5.2.1 Modeling techniques

Each processor is modeled at the micro-operation or register-transfer levels. While the models explicitly describe each interaction between the datapath components, the architectures are not modelled at the gate level. The aim is to describe the architectures to characterise their performance without binding to particular implementations.

The simulator kernel provides the basic functionality necessary to manage an event queue, while a small library of code provides mechanisms for instantiating and interconnecting components.

At the highest level each processor model is described in a structural form, detailing the interconnections between datapath components. The components themselves describe functionality at a micro-operation level. Figure 5.2 provides an example in pseudo-code of how a typical component, in this case a simplified operand fetch stage, is modelled.

Events generated by other components are managed by a central event queue. When events reach the end of the queue they cause the relevant component function to be called with the appropriate state and input event. The component then processes the input event, and if necessary produces one or more output events in response. For example, the arrival of an `INSTRUCTION` event causes others to be produced to fetch each of its operands. These events would then subsequently be handled by the register file and forwarded result queue components. Delays may be inserted by modifying an output event's time-stamp, which causes the event to be inserted in the event queue for handling at the appropriate time. The implementation uses a single thread of execution requiring that each component return control back to the kernel after processing their input event.

```
// create new event and add it to the output event list
void new-event (output-event-list, destination, event)

// Example component description
void operand-fetch (component-state, event-in, events-out)
{
  cs=component-state

  switch (event-in):
  {
    case INSTRUCTION:
      // Handle receipt of new instruction
      instr=event-in->instr-data
      for 1..instr.num-operands do
        // send request to source of operand
        // register file or forwarding queue
        new-event (events-out, instr.op-source[operandn], REQUEST)

      cs->outstanding-operands=instr.num-operands
      cs->instr=instr

    case DATA:
      // Data received from register file
      cs->outstanding-operands--


    case FWD-DATA:
      // Data received from forwarded result queue
      cs->outstanding-operands--


    case ACK:
      // Acknowledge from output received
      cs->output-ready=TRUE

    default:
      Error!
  }

  // If operands have been fetched and all outputs are
  // ready to receive new data, send instruction to FU
  if ((cs->outstanding-operands==0)&&
    (cs->output-ready)&&
    (cs->instr!=NULL)) {
    new-event (events-out, cs->fu-output, cs->instr)
    cs->output-ready=FALSE
    cs->instr=NULL
  }
}
```

Figure 5.2: Pseudo-code example of a simple operand fetch model

Benchmark Name	**Description**
compress	Standard UNIX text compression (input is 5k of random text)
cjpeg	GIF to JPEG image compression (input is 40x26 GIF image)
fgrep	Search a file for a character string (input is 100k text file)
gcc	GNU C Compiler (input is bubblesort.i)
go	Go game
perl	Perl interpreter (input is script to find prime numbers)
xlisp	LISP Interpreter (input is queens program)

Table 5.1: Descriptions of benchmark set and their inputs

Benchmark Name	**Trace Length**	**BB size**	**Result fanout**	**Operand per instr.**	**Dcache miss rate**
compress	450K	7.49	1.84	1.28	1.32
cjpeg	530K	7.12	1.80	1.40	0.66
fgrep	1.5M	4.20	1.83	1.29	0.33
gcc	1.5M	4.67	1.72	1.20	1.47
go	1.5M	4.82	1.98	1.29	4.04
perl	795K	5.19	1.68	1.14	0.44
xlisp	600K	5.23	1.69	1.16	0.73

Table 5.2: Benchmark statistics. BB size - Average basic block size, Operand per instr - Average number of register operands per instruction (excluding branches).

5.2.2 Benchmarks

The benchmarks used for the experiments are listed in Table 5.1, which are similar to those found in the SPEC CINT95 benchmark suite. A detailed description of the benchmarks may be found at [135]. In contrast to the input data required to run the full SPEC benchmarks, the results presented are for shorter inputs that reduce simulation times to realistic levels. Benchmarks whose traces were still deemed too long were truncated to 1.5M instructions. An overview of some of the characteristics of each benchmark is given in Table 5.2, together with dynamic instruction frequencies in Table 5.3. The benchmarks provide a typical sample of compute intensive integer benchmarks which contain modest levels of ILP.

The benchmarks were compiled using GCC (version 2.8.1), with the optimisation level set to three. Binaries were statically linked to ensure that libraries were also instrumented and traced. A final requirement was that GCC was instructed to use a single register window and not generate `SAVE` and `RESTORE` instructions, which was necessary as the instruction traces were collected from a SPARC processor which exploits a register window mechanism.

Benchmark Name	**Loads**	**Stores**	**Add/ Sub**	**Logic**	**Shift/ Mult**	**Branch**	**Other**
compress	20.31	14.59	27.53	5.61	7.95	13.35	10.66
cjpeg	18.31	9.49	36.11	3.83	12.39	14.04	5.83
fgrep	23.37	7.70	35.37	2.04	0.70	23.79	7.03
gcc	14.94	10.62	26.92	5.00	7.08	21.41	14.03
go	12.21	9.03	31.14	2.27	9.47	20.76	15.12
perl	21.04	11.57	22.88	5.12	3.19	19.27	16.93
xlisp	18.36	15.17	24.85	4.34	3.06	19.13	15.09

Table 5.3: Dynamic Instruction Frequencies (%). The category "other", includes set/move and clear instructions

5.2.3 The models

Results have been collected from four different processor models. Firstly, the synchronous superscalar processor detailed at the beginning of Chapter 3 was characterised. Secondly, results generated from an asynchronous version of this architecture are presented. This model again includes full dynamic scheduling and data-forwarding capabilities, which are implemented using the techniques described in Chapter 3. The third model describes a queue-based architecture, where dispatch is in-order at each functional unit. Finally, instruction compounding is explored using models of the datapaths detailed in Chapter 4. Results exploiting both static and dynamic compounding techniques are presented.

Each model uses a similar address disambiguation and memory access model. Where dynamic scheduling permits, loads are able to be executed out of order with respect to independent store instructions – although no store to load forwarding is implemented. Address disambiguation is performed speculatively and always returns the correct answer. This corresponds to the existence of a perfect load wait table. This removes the possibility of load instructions being dispatched before an earlier store accessing the same memory location has completed; in reality such situations would occur and raise an exception. The final detail describes the behaviour of the dispatch logic when an instruction uses the result of a load instruction. In this case no cache hit/miss prediction is made available, and all instructions which require the result of a load are dispatched speculatively assuming the load will be a hit. If this is not the case, and the load in fact misses, then the instruction will be stalled in its operand fetch stage until the data becomes available. This behaviour is the same in both the synchronous and asynchronous processors and may result in functional units remaining idle while cache misses are being serviced.

Also common to each model is use of register renaming and the existence of

Parameter	Value
Fetch Width	4 instructions/cycle
Functional Units	2 Memory units, 2 ALUs, 1 Complex ALU (mult/div/shift)
Branch Prediction	Perfect
Instruction Window	0-80
Physical Registers	80
L1 I-cache	Infinite, 1 cycle latency
L1 D-cache	64k, 2 cycle latency, 2-way, LRU, 32-byte blocks
L2 D-cache	Infinite, 12 cycle latency

Table 5.4: Common Parameters

a reorder buffer. No mechanism is included to detect and enforce carry/borrow dependencies in any of the models, but would be simple to add [114]. This type of dependency was very rare in the benchmarks, and any impact on performance would be undetectable.

A list of parameters common to each of the models is provided in Table 5.4. The choice of a four-way superscalar architecture was made as it is only at these levels of parallelism, for the given set of benchmarks, that dynamic scheduling and data-forwarding become critical. While the performance of two-way superscalar models tends to saturate, the results provided by the four-way models show clear trends indicating different abilities of the architectures to expose ILP at run-time.

Delays used in the asynchronous models are listed in Table 5.5. Note that in the asynchronous case, cache and register-file accesses (for the 2-port per FU instance) are same as those used in the synchronous model. A model of register file access times was provided using a modified version of the Cacti cache access model [71], which was also used to provide results for [107]. SPICE simulations of transistor netlists provided an approximation to the delays of arbiters, although metastability is not modelled. In practice many of the models are only sensitive to a few critical delays, such as register-file read/write delays and data-cache latency. The differences in performance are also heavily influenced by an architecture's ability to perform dynamic scheduling or the restrictions on performing data-forwarding operations.

More specific details of each model are provided at the beginning of each of the following sections where appropriate.

Delay Description	**Rel. delay**	**Abs. delay (pS)**
Register Read/Write		
2 read ports per FU	100	2891
1 read port per FU	78	2245
Functional units		
ALU (add/shift)	50	1445
Set/Move/Clear	0	0
Logical	20	578
Memory (load/store)	100	2891
Typical FIFO buffer throughput	4 items/cycle	
Queued		
Write to fwding reg.	7	200
Data Forwarding delay	6.4+3.5*reg	185+100*reg
Async. Dispatch		
Forward req. arbiter		
Req. to grant	17.5	505
Cycle time	28.8	832
Forwarding		
Write to fwding reg.	7	200
Data Forwarding delay	6.4+10.5*reg	185+300*reg
Dispatch Buffer		
Write instruction	34.5	1000
Max. dispatch rate	2 instr/port/cycle	
Data write/wakeup	2 instr/port/cycle	
Min. wakeup to dispatch delay	100	2891
Reg Scoreboard Access	50	1445
Compounding		
Dispatch Buffers		
Data write Latency	10.4	300
Dispatch Latency	50	1445
Busy-bit table access	34.5	1000
Forwarding		
Fwding req. arbiter		
Req. to grant	17.5	505
Cycle time	28.8	832
Forward data delay	10.4	300

Table 5.5: Overview of delays. Rel. delays - are shown as a percentage of the synchronous clock period. Data forwarding delays exclude any result buffering delays

5.3 Synchronous Processor

The synchronous processor model describes the architecture introduced in Section 3.2. The processor's dispatch buffer may be configured to operate either as a centralised buffer storing instructions of all types, or one in which the dispatch buffer is divided amongst each functional unit (distributed). In the case of the synchronous model a completely unrestricted data forwarding mechanism is implemented. To explore the impact of data forwarding this may be artificially constrained, either to within basic blocks or switched off completely. The simulations explore the impact of each of these parameters, while also sweeping across a range of dispatch buffer sizes.

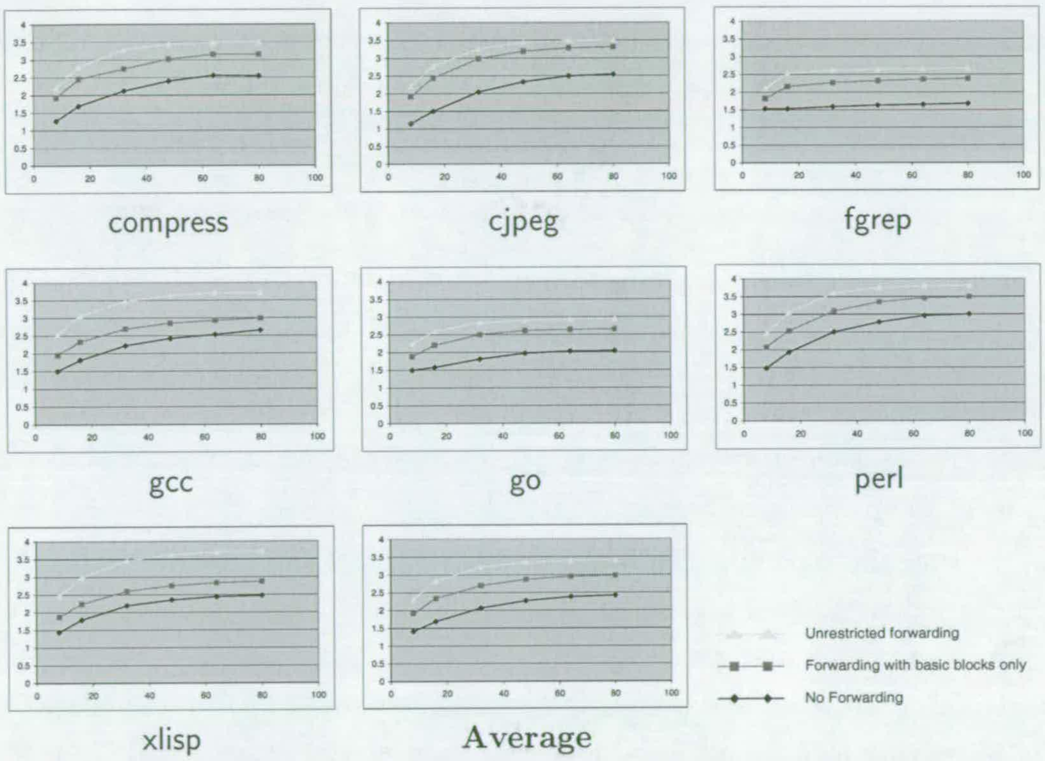


Figure 5.3: Impact on performance of dispatch buffer size and forwarding policy, for a centralised dispatch buffer. Graphs show IPC vs. buffer size. (synchronous)

Performance results for the experiments are shown in Figures 5.3 and 5.4 for centralised and distributed dispatch buffer configurations, respectively. On average the difference in performance, when forwarding is unrestricted, between the centralised and distributed cases is around 6.5%. These results are summarised in Figure 5.5. The slightly lower performance of the distributed window is a result of the fact that on average it can only take advantage of a subset of all of the

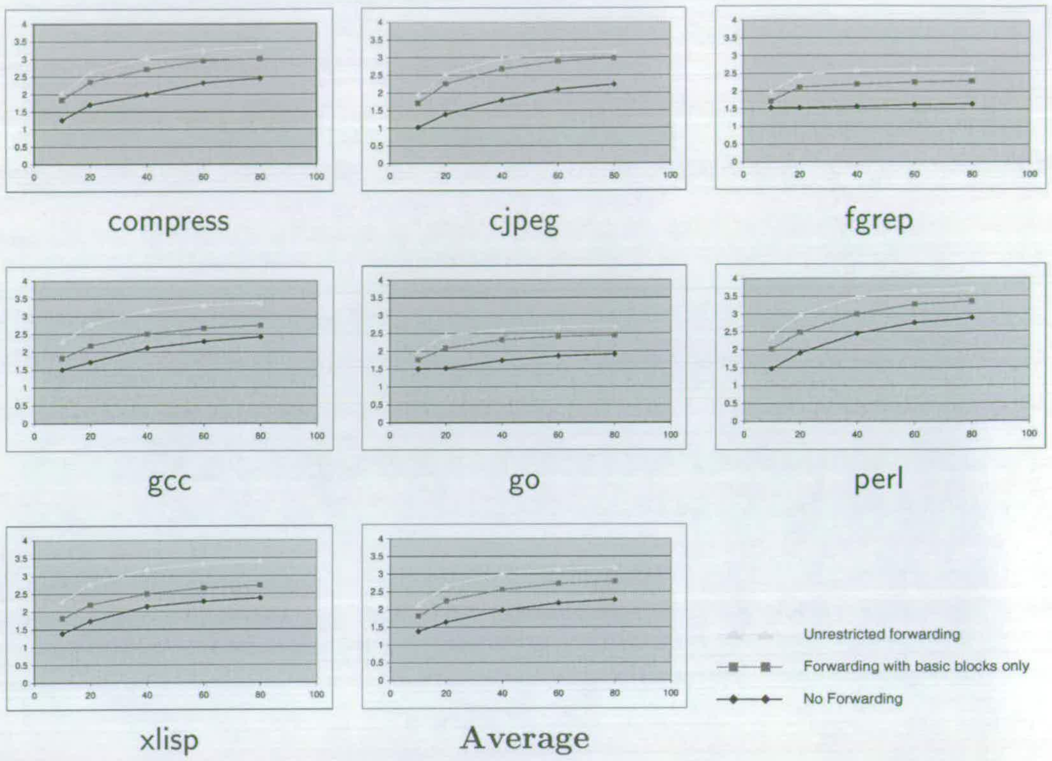


Figure 5.4: Impact on performance of dispatch buffer size and forwarding policy, for a distributed synchronous dispatch buffer. Graphs show IPC vs. buffer size. (synchronous)

available reservation stations. This is because instruction issue must stall when any one of the dispatch buffers becomes full.

A more significant impact on the performance comes from restricting the data-forwarding capabilities of the model. Results for the case when the dispatch buffer size is set to a maximum for a distributed window are summarised in Figure 5.6. A performance drop of around 30% is incurred for removing forwarding altogether, while restricting forwarding to basic blocks reduces the measured IPC on average by 13%, with similar reductions in performance for the case of a centralised window. The number of results forwarded as a percentage of the total number of operands is shown in Figure 5.7. It can be seen that on average around 40% of operands are forwarded for the case where forwarding is unrestricted and the dispatch buffer size is set to its maximum. Restricting forwarding to within basic blocks reduces this figure to just over 25%.

The performance benefits of increasing the size of the dispatch buffer are clear for up to 40 entries; after this the gains in performance from additional reservation stations are far more modest. Overall, the difference between the performance at the smallest window size of 10, and the largest one of 80, is around 35% for both centralised and distributed windows.

The results provide a base case for comparing the performance of each of the different architectures. By reproducing well known results [62, 68] a basic validation of the simulation environment has also been performed.

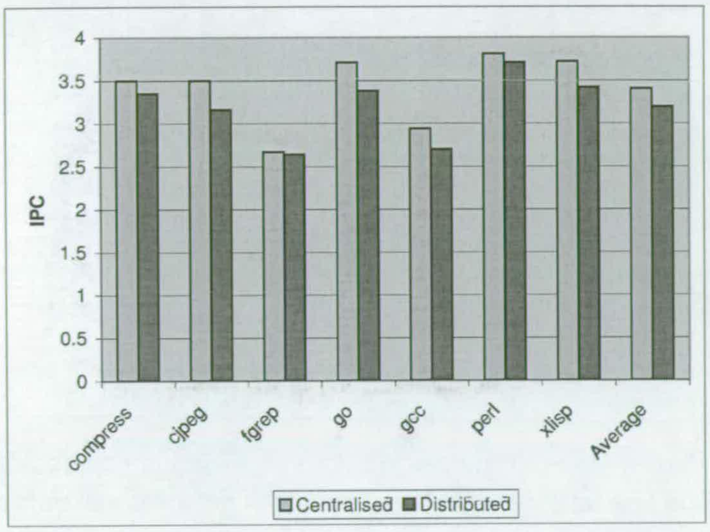


Figure 5.5: A comparison of centralised and distributed dispatch windows. The dispatch buffer's size is set to maximum and forwarding is unrestricted. (synchronous)

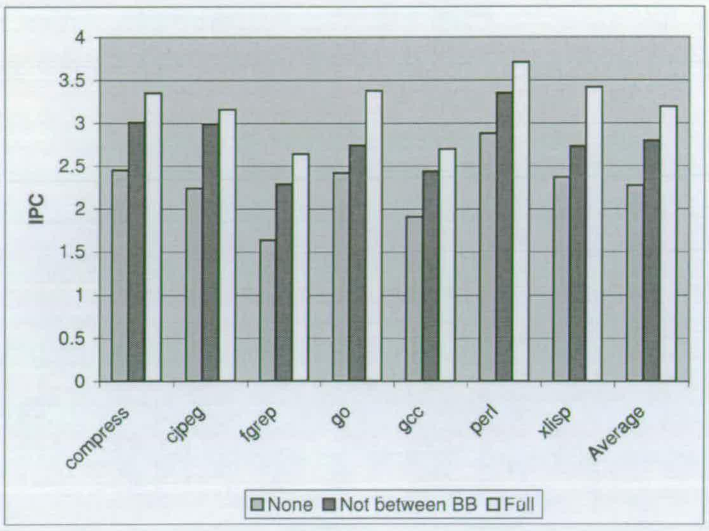


Figure 5.6: Performance comparison of a synchronous superscalar processor with full, restricted and no data-forwarding. The dispatch buffer size is set to maximum and distributed. (synchronous)

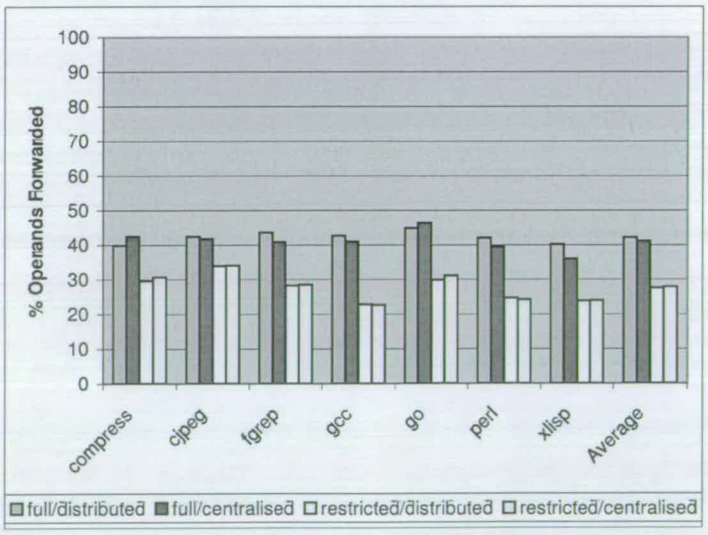


Figure 5.7: Levels of forwarding for cases where data-forwarding is restricted and unrestricted. Results are shown for both distributed and centralised dispatch buffers. Buffer size is set to maximum. (synchronous)

5.3.1 Queue-based Asynchronous Processor

The queue-based model was first described in Section 3.4.1. As described previously, dispatch is in order at each functional unit. Data forwarding is implemented by detecting opportunities to forward results during instruction decode. The number of results buffered at the output of each functional unit may be specified as a configuration parameter. The architecture does not implement the simplifications described in Section 3.4.2 and supports full out-of-order write-back for maximising performance.

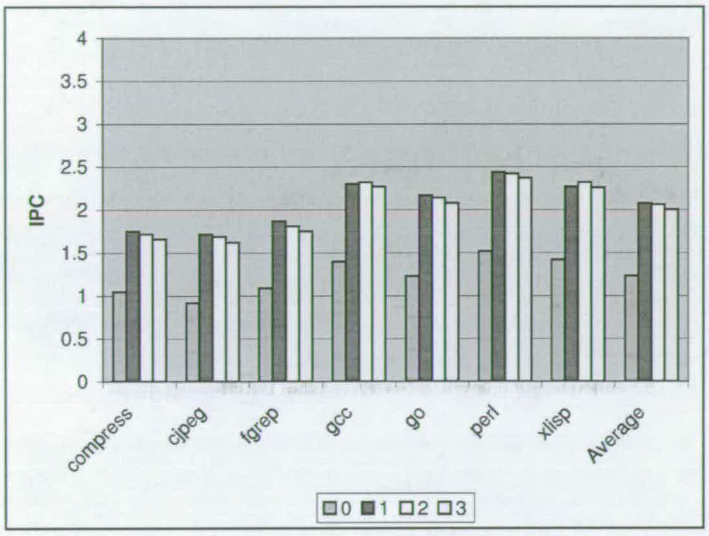


Figure 5.8: Impact on performance of number of forwarding registers. Instruction queues are 16 entries long at each functional unit(queued)

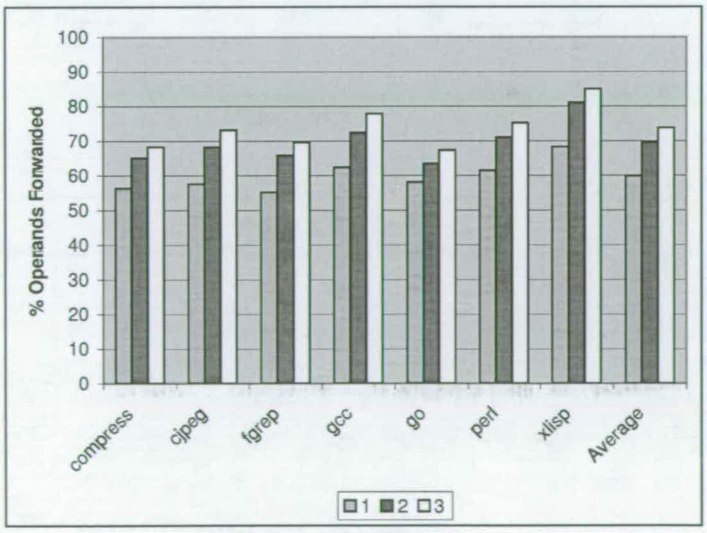


Figure 5.9: Levels of forwarding for 1,2 and 3 forwarding registers per FU. Instruction queues are 16 entries long at each functional unit(queued)

The first results explore the impact of increasing the number of forwarding registers at each functional unit. This allows more results to be buffered at each functional unit, which creates a greater scope for data forwarding. Figure 5.8 shows results for 0,1,2, and 3 forwarding registers. Although the differences between 1,2 and 3 registers are small, removing the possibility of forwarding altogether results in a 41% drop in average IPC [1]. This percentage drop in performance is larger than that seen previously in the synchronous model. This is a result of the limitations in the queue-based model's dynamic scheduling scheme. Unlike schemes which allow full out-of-order execution, the order in which instructions must be dispatched at each FU is fixed - providing less opportunities to adapt to the increased operand fetch latencies. It can also be seen that increasing the number of forwarding registers beyond 1 does not provide any real performance advantage. One explanation is that levels of forwarding, as shown in Figure 5.9, are already high for a single forwarding register. Adding additional registers both increases the delays in accessing the forwarding registers and increases the amount of traffic on the forwarding network, as a result any additional forwarding of data has little overall performance benefit.

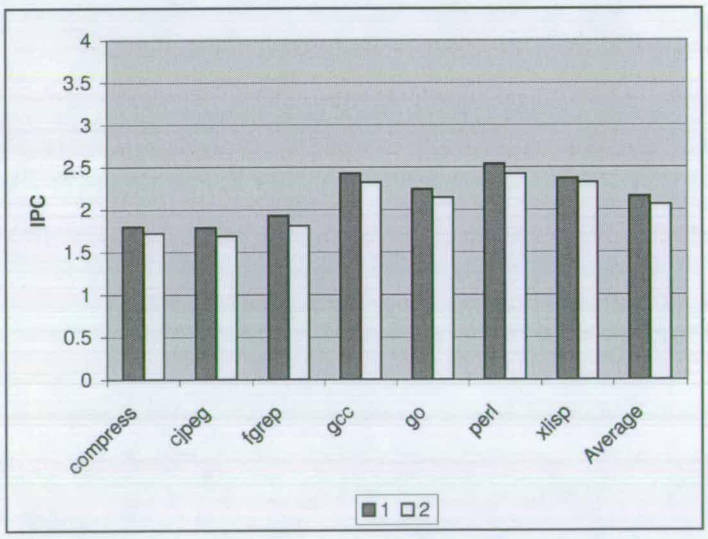
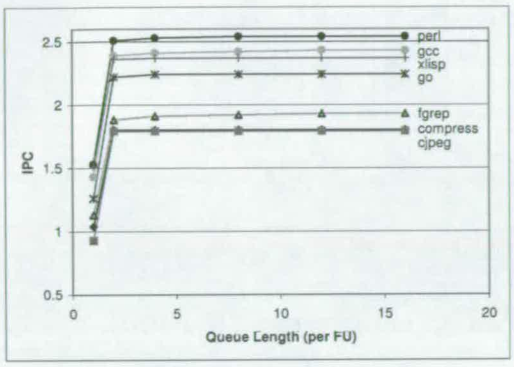


Figure 5.10: Impact on performance of number of register read ports per FU. Instruction queues are 16 entries long at each functional unit(queued)

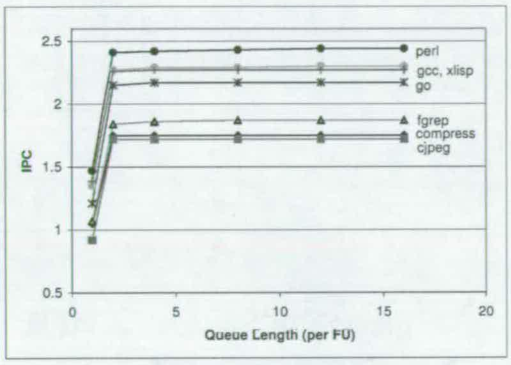
The second configuration examined the reduction in the number of register read ports available to each functional unit. With forwarding levels around 60%, it is reasonable to assume that in the majority of cases only a single register read port will be required for each instruction. Results for the single and dual

[1]For the asynchronous models IPC is calculated by measuring execution time and dividing it by the clock period of the synchronous processor.

register read port cases are shown in Figure 5.10. Here it can be seen that a small increase (5%) in performance is produced when the number of register read ports are halved. This is a result of the asynchronous architecture's ability to exploit a lower register read/write latency when only a single operand must be fetched from the register file. Although the penalty of sequential register reads has to be incurred when two or more operands must be fetched from the register-file, on average, the performance is improved as the complexity of the register-file is reduced significantly.



Dual read ports



Single read port

Figure 5.11: Impact on performance of instruction queue length. The leftmost graph shows case where two register read ports are available to each FU, while the other is shows the case where only a single port is available. (queued)

Various instruction queue lengths were also explored, and the results of these experiments are shown in Figure 5.11. It can be seen that there is a sharp increase from 1 to 2 FIFO entries per FU, after which the IPC levels off quickly with little increase after 4 FIFO entries per FU. This behaviour is similar irrespective of the number of register read ports.

The final sets of experiments explore the impact of the register file access time and functional unit latencies. Figure 5.12 contrasts the performance of each benchmark when the FUs have latencies equivalent to a full clock period or the reduced latencies listed previously. On average, the reduced functional unit latencies provide almost a 20% increase in performance (for the single register port case). Reducing the register file access time by 50% produces on average a 6% and 10% increase in IPC, for the single and dual port configurations respectively. The fixed dispatch schedule at each functional unit and the high levels of data-forwarding mean that reducing the register-file access time does not have a dramatic effect on overall performance.

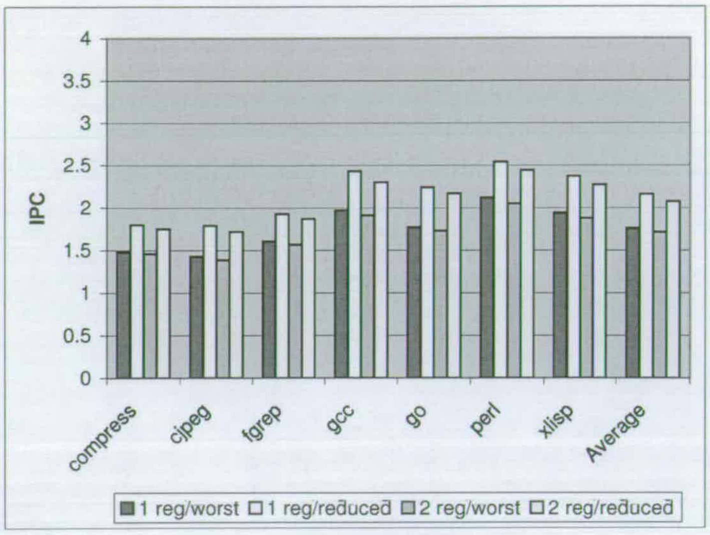
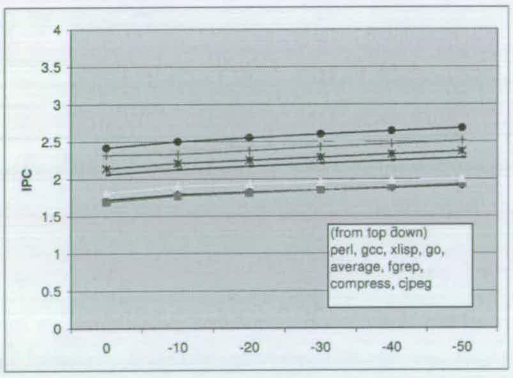
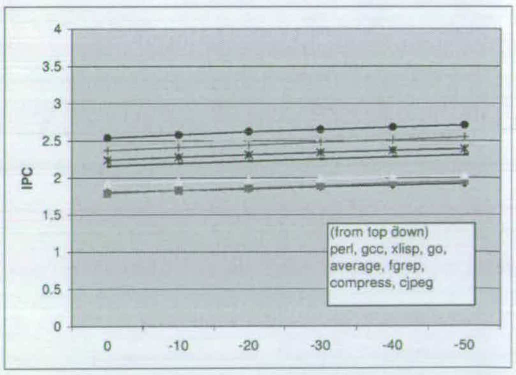


Figure 5.12: Impact of FU latencies on performance. (queued)



Dual read ports per FU



Single read port per FU

Figure 5.13: Impact of reducing register file access time on IPC. Reduction ranges from 0-50% of original delay. (queued)

5.3.2 An asynchronous processor with out-of-order dispatch

This section presents simulation results for an asynchronous superscalar processor which supports both out-of-order dispatch and data forwarding. The techniques used to implement such a processor are described at length in sections 3.3 and 3.4. The model uses a centralised dispatch buffer and places no restrictions on which instructions may request forwarded data. As in the queue-based model the data-forwarding scheme may be configured with different numbers of forwarding registers at each functional unit. The dispatch buffer itself, models the wake-up and selection processes using fixed delays. While ready instructions can be dispatched at the rate of 2 per synchronous clock period, the minimum time for the wake-up, selection and dispatch of an instruction takes the equivalent of a full synchronous clock period. The aim of the experiments is not to investigate any one particular implementation of the dispatch buffer, but to ensure that the arbitrated access to the dispatch buffer and the limited data-forwarding mechanism do not have a detrimental impact on overall performance.

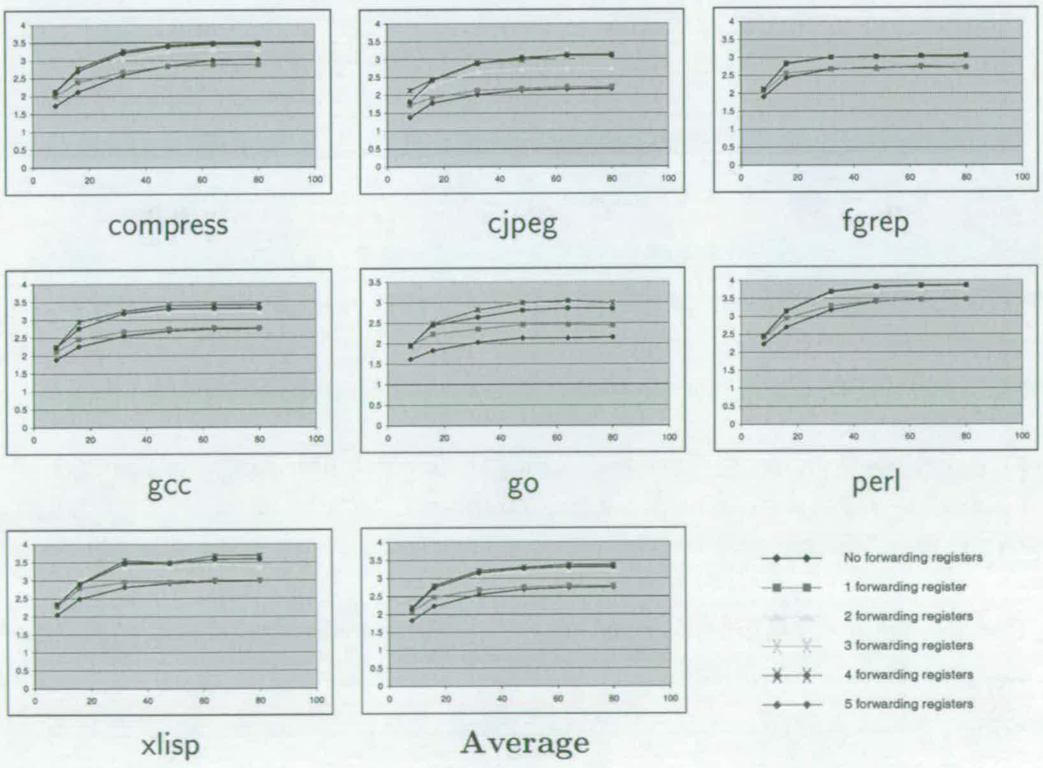


Figure 5.14: Impact on performance of dispatch buffer size and number of forwarding registers (no forwarding counters are present). Graph shows IPC vs. buffer size. (async. dispatch)

The first set of results (see Figure 5.14) explore the effect of the size of the dispatch buffer and the number of forwarding registers on performance. Forwarding is initially not supported by source counters (see Section 3.4.3.1). The best configuration is one with 4 forwarding registers per functional unit. The difference between this configuration and one where no forwarding is performed is around 20%. The penalty of removing data-forwarding altogether is lower than in the synchronous case, which showed a 30% drop, this is a result of the asynchronous processor's ability to exploit reduced functional unit latencies. As expected, the trend of the curves generated by sweeping a range of dispatch buffer sizes are very similar to those generated from the synchronous processor.

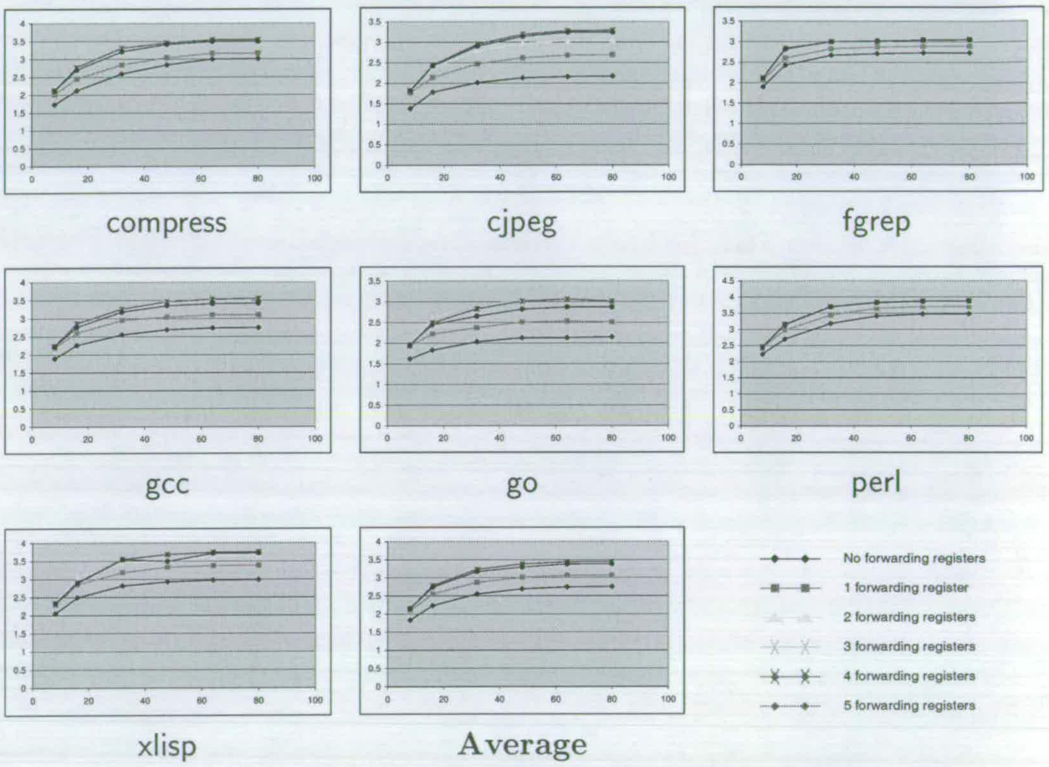


Figure 5.15: Impact on performance of dispatch buffer size and number of forwarding registers (with forwarding counters). Graphs show IPC vs. buffer size (async. dispatch)

The previous experiments were repeated for the case where source counters are exploited (see Section 3.4.3.1). The counters aim to improve performance by reducing the number of forwarding operations which must be aborted. Once forwarding is initiated at a particular reservation station, subsequent update operations from the same source functional unit reduce the corresponding operand's source counter. If the counter reaches zero before the instruction is dispatched, then the forwarding operation is cancelled. Source counters are initialised to the same value as the number of forwarding registers at each functional unit. A complete set of results where source counters are exploited is shown in Figure 5.15. A comparison of average performance for a range of different forwarding register/source counter configurations is presented in Figure 5.16. Here the highest performing schemes, 4 forwarding registers without counters and 3 with counters, are shown against results where no forwarding and a single forwarding register were configured. While the highest performing scheme is one in which counters are employed, adding a forwarding register and removing the counters only incurs a performance penalty of just over 2%.

Increasing the number of forwarding registers increases the number of operands which may be forwarded, while reducing the number of forwarding operations which must be cancelled. The use of source counters also reduces the number of cancelled forwarding operations. Results illustrating these relationships are shown in Figure 5.17. At 3 forwarding registers the level of forwarding activity is comparable to the results provided by the synchronous processor model.

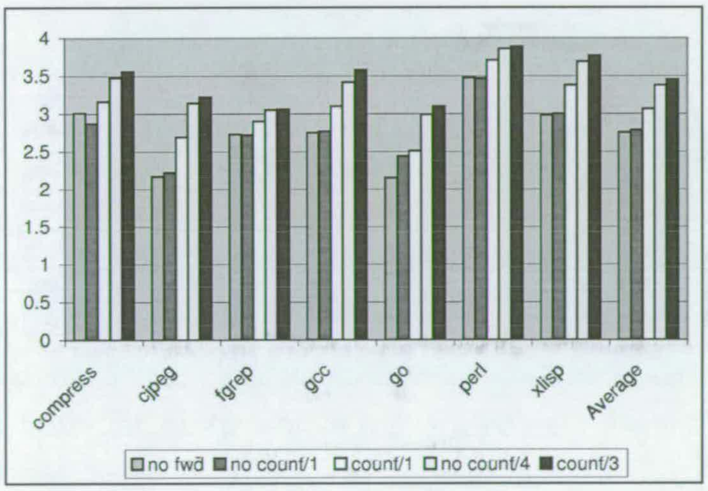


Figure 5.16: A comparison of a number of different forwarding register/counter configurations. 4 and 3 forwarding registers provide the best performance for the cases where counters are not used and used respectively. (async. dispatch)

Results where also generated to measure the impact of reducing functional unit latencies on performance. The results are summarised in Figure 5.18. On average a 7% performance improvement was possible through reducing functional unit latencies from a fixed one cycle delay to those listed in section 5.2.3. This is smaller than the 20% increase possible in the queued-model. This is a result of the model's superior dynamic scheduling capabilities and corresponding reduced sensitivity to the performance of sequential operations.

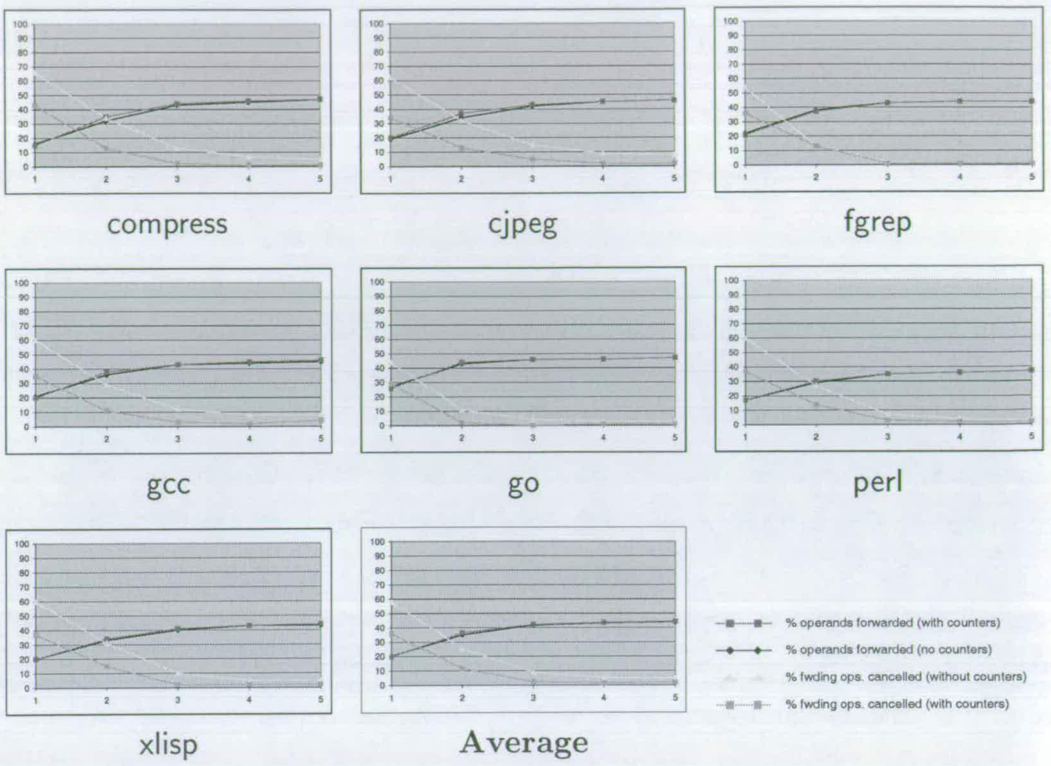


Figure 5.17: Percentage of operands forwarded and percentage of forwarding operations cancelled. Results are shown for different numbers of forwarding registers and the cases where forwarding counters are and are not employed. (async. dispatch)

The use of a single register read port and the impact of a reduction in register access time is explored in Figure 5.19. Again, asynchrony allows the number of register read ports to be reduced without any significant change in average performance. Register file access times in general do not effect performance in any significant way, except in the case of the CJPEG benchmark.

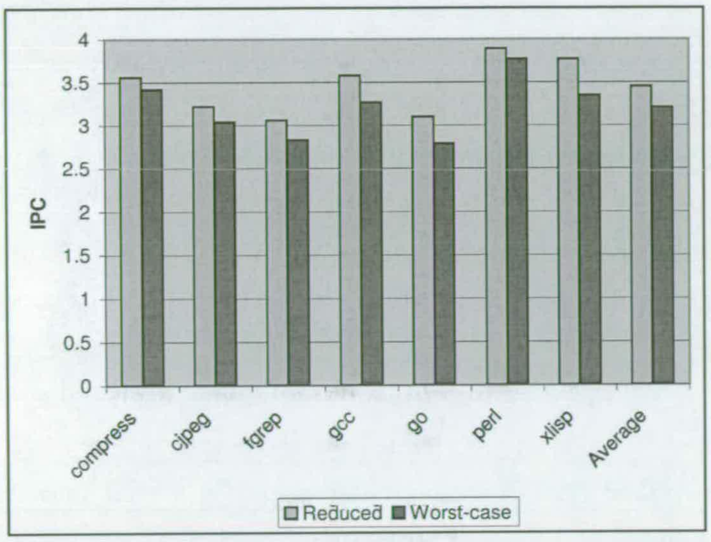
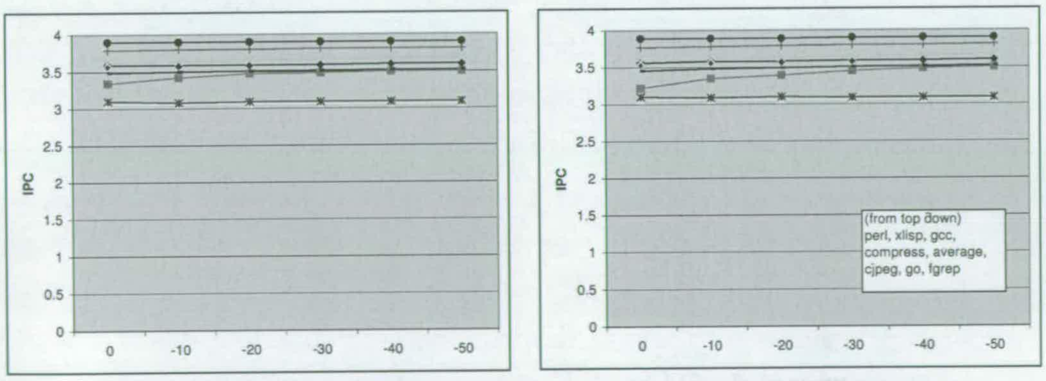


Figure 5.18: Impact of worst-case functional unit delays on best performing configuration: 3 forwarding registers with forwarding counters. (async. dispatch)



Dual read ports per FU Single read port per FU

Figure 5.19: Impact of reducing register file access time on IPC. Reduction ranges from 0-50% of original delay. (async. dispatch)

5.3.3 An asynchronous compounded instruction processor

This section provides results for a processor that compounds instructions. The compounds are produced using a simple graph partitioning algorithm, compounding opportunities were selected at random and no attempt was made to optimise the final compounded instruction schedule. A complete description of the compounding processor is provided in section 4.3. The dynamic compounding mechanism is also modelled, which permits data-forwarding across basic block boundaries. In this case no delay was imposed on the construction of the compounds or the reading of compounding information from the forwarding table. The results were simply used to explore the likely performance advantages of allowing compounds to span basic block boundaries. The results were collected for a model which includes two busy-bit read queues for each functional unit. In the case of memory units an additional busy-bit read queue is added and used exclusively for store instructions.

We first explore the relationship between the compounding dispatch buffer size and performance. Results are included for two different configurations of the dispatch buffer. The first one where instructions at each functional unit are stored in a single dispatch buffer; and the second, in which instructions are stored in one of two buffers at each functional unit depending on their position in a compound (split buffers). In the case of split buffers, those instructions which will receive forwarding requests are stored in a separate buffer. In an instruction compounding processor only those instructions which follow the first instruction in the compound are eligible to receive forwarded data. Hence we will call the two buffers at each functional unit the head and tail buffers. It is important to note that these configurations are local modifications at each functional unit, it is never the case that a single monolithic dispatch buffer serves the functional units.

Figure 5.20 shows the configuration in the split buffer case. The equivalent diagram for the unified buffer case has been described previously in Section 4.3.2. Instructions are written into one of the two buffers depending on whether they can receive forwarded data or not. Subsequent busy-bit read micro-operations are tagged with both the buffer (head or tail) and particular buffer entry that will be updated when the busy-bit read completes. Splitting the dispatch buffer in this way simplifies the implementation of the head buffer by removing the need to support the wake-up logic associated with forwarding requests. A modest increase in concurrency is also possible as we can now perform dispatch request arbitration for the two buffers in parallel. An additional arbiter is required to arbitrate between dispatch requests from each buffer.

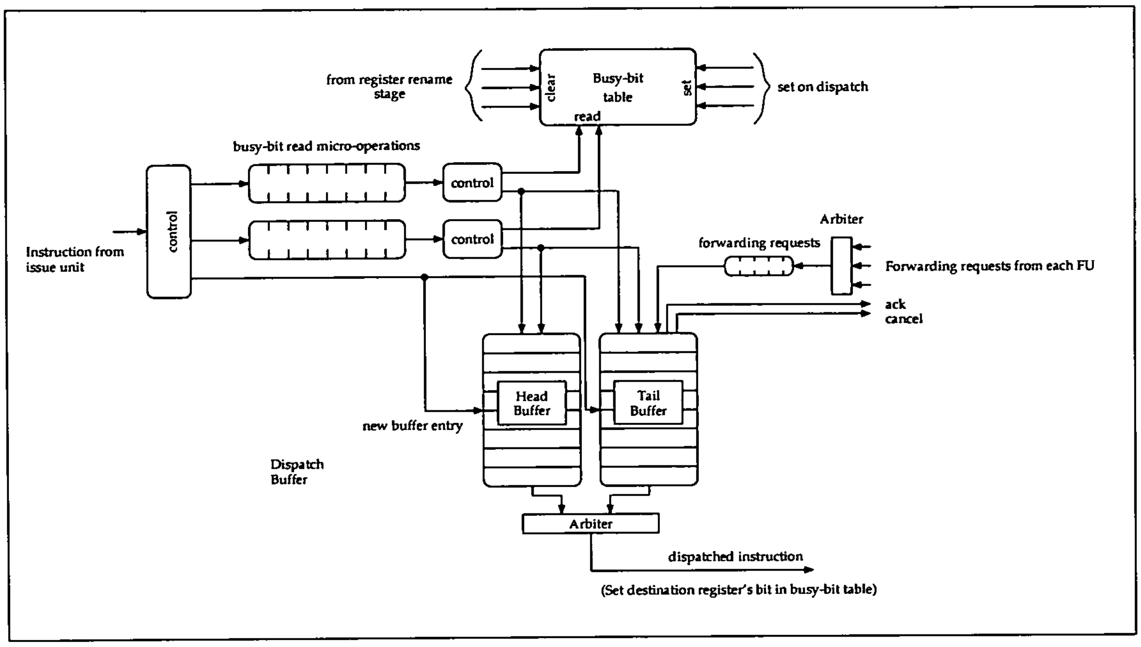


Figure 5.20: The split dispatch buffer configuration

The results in Figure 5.21 show that for larger buffer sizes the performance difference between the two configurations for an equal number of reservation stations is minimal. Due to the slight improvement in complexity and performance offered by the split buffer configuration all subsequent results in this section will be based on a model configured with split buffers. For all the results shown in Figure 5.21, the delays associated with the buffer have remained constant, although a smaller buffer would in reality reduce access delays, this effect had a very small impact on performance.

The differences in performance between statically and dynamically compounding programs and the use of a single or dual register read ports per functional unit, is summarised in Figure 5.22. Figure 5.23 records the levels of data forwarding for the same model configurations. While dynamic compounding has greater scope for creating opportunities to forward data, overall levels of result forwarding are very similar in all the configurations. This can be explained by examining the graph presented in Figure 5.24, which records the percentage of forwarding operations initiated but subsequently cancelled. In the case where dynamic compounding is employed, it can be seen that cancellation levels are significantly higher than those when compounds are constructed statically. Even though this is the case, the overall performance benefits from the ability to construct compounds dynamically, without restricting forwarding to within basic blocks.

arbitrate between dispatch requests from each buffer.

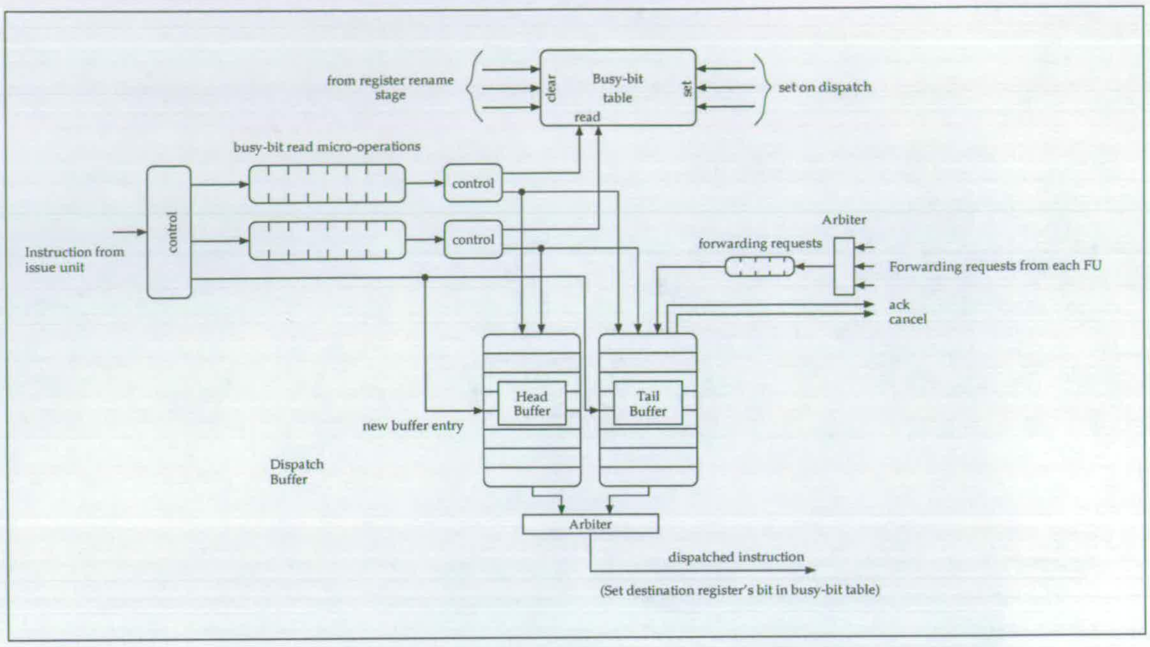


Figure 5.20: The split dispatch buffer configuration

The results in Figure 5.21 show that for larger buffer sizes the performance difference between the two configurations for an equal number of reservation stations is minimal. Due to the slight improvement in complexity and performance offered by the split buffer configuration all subsequent results in this section will be based on a model configured with split buffers. For all the results shown in Figure 5.21, the delays associated with the buffer have remained constant, although a smaller buffer would in reality reduce access delays, this effect had a very small impact on performance.

The differences in performance between statically and dynamically compounding programs and the use of a single or dual register read ports per functional unit, is summarised in Figure 5.22. Figure 5.23 records the levels of data forwarding for the same model configurations. While dynamic compounding has greater scope for creating opportunities to forward data, overall levels of result forwarding are very similar in all the configurations. This can be explained by examining the graph presented in Figure 5.24, which records the percentage of forwarding operations initiated but subsequently cancelled. In the case where dynamic compounding is employed, it can be seen that cancellation levels are significantly higher than those when compounds are constructed statically. Even though this is the case, the overall performance benefits from the ability to construct compounds dynamically, without restricting forwarding to within basic blocks.

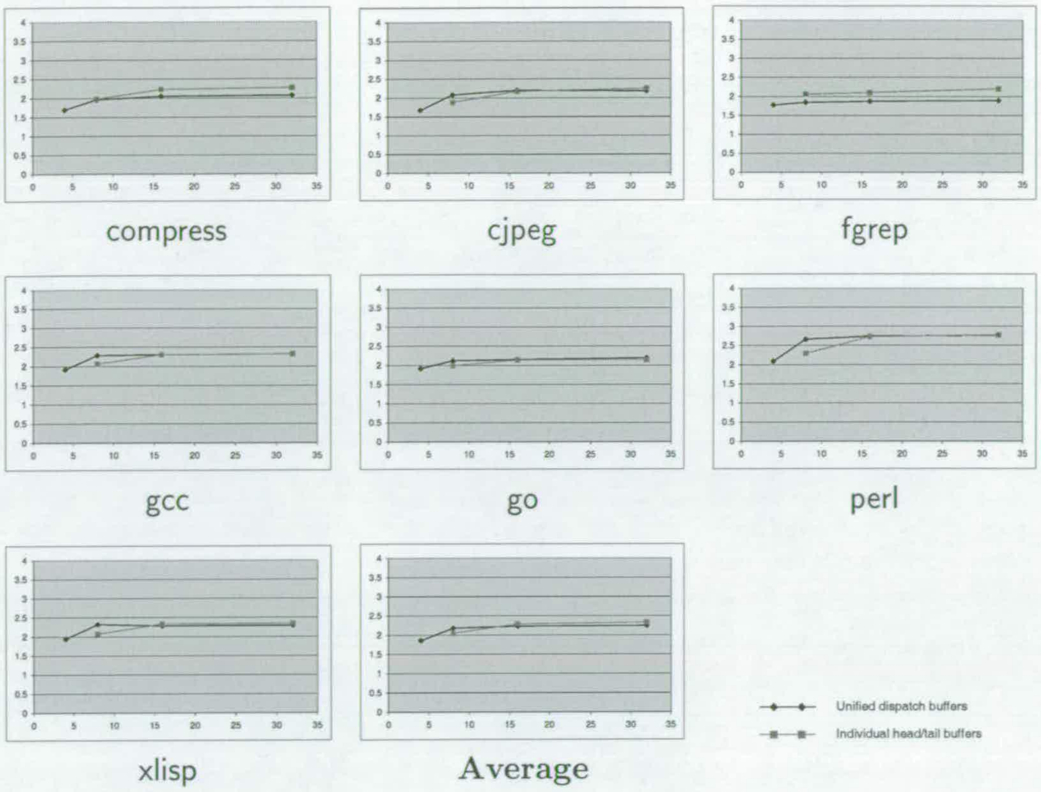


Figure 5.21: Impact on performance of dispatch buffer size and configuration. Graphs shows IPC vs. buffer size, for unified and split buffer configurations. (compounding)

Results illustrating the impact of register file access times are presented in Figure 5.25. It can be seen that reducing register file access times, in the case of a compounding architecture, has a significant impact on overall performance. A reduction of 50% in register file access time produces around a 20% improvement in the performance of all the configurations. The larger impact of register file access time is most likely a result of the lower levels of data-forwarding, when compared to the previous two models. In the compounding case only just over 20% of operands are forwarded, this compares to around 40% in the case of the asynchronous dispatch buffer and 60% for the queue-based model.

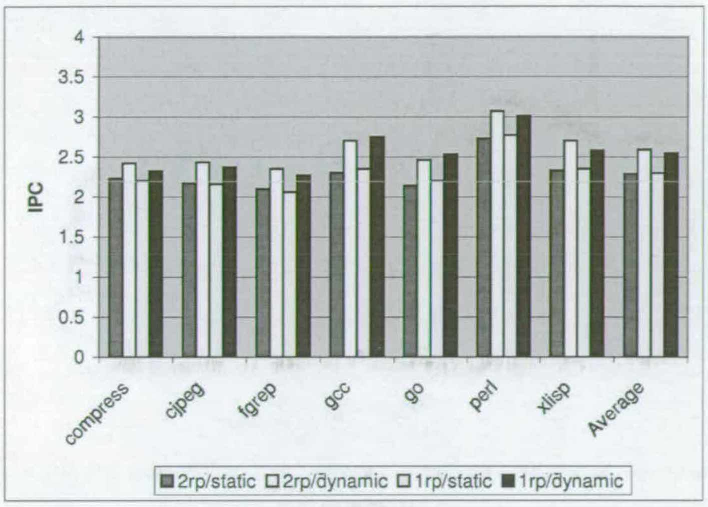


Figure 5.22: Performance of instruction compounding datapaths. Results are shown for both static and dynamic cases, with single and dual register read ports per FU. (comp)

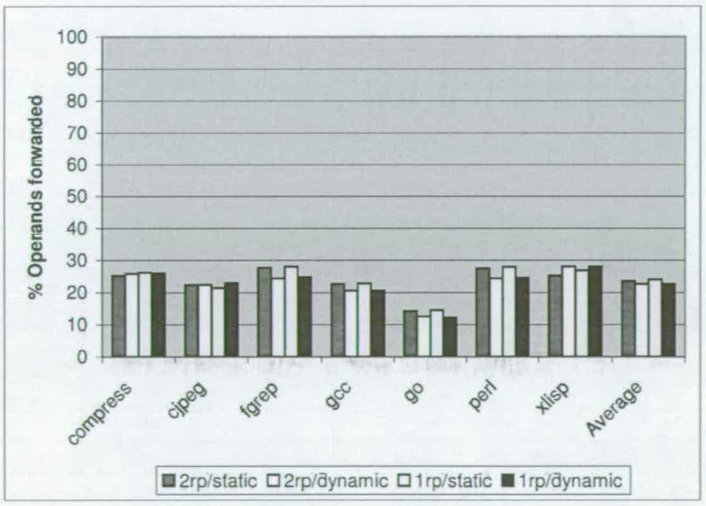


Figure 5.23: Levels of forwarding in compounding datapaths. Results are shown for both static and dynamic cases, with single and dual register read ports per FU. (comp)

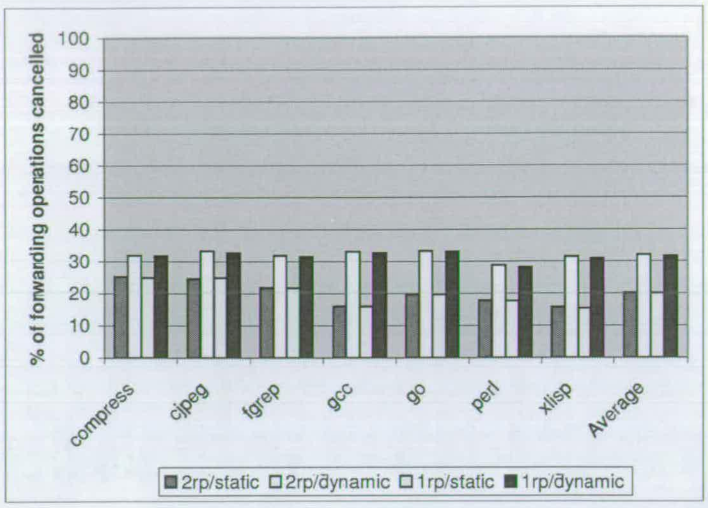
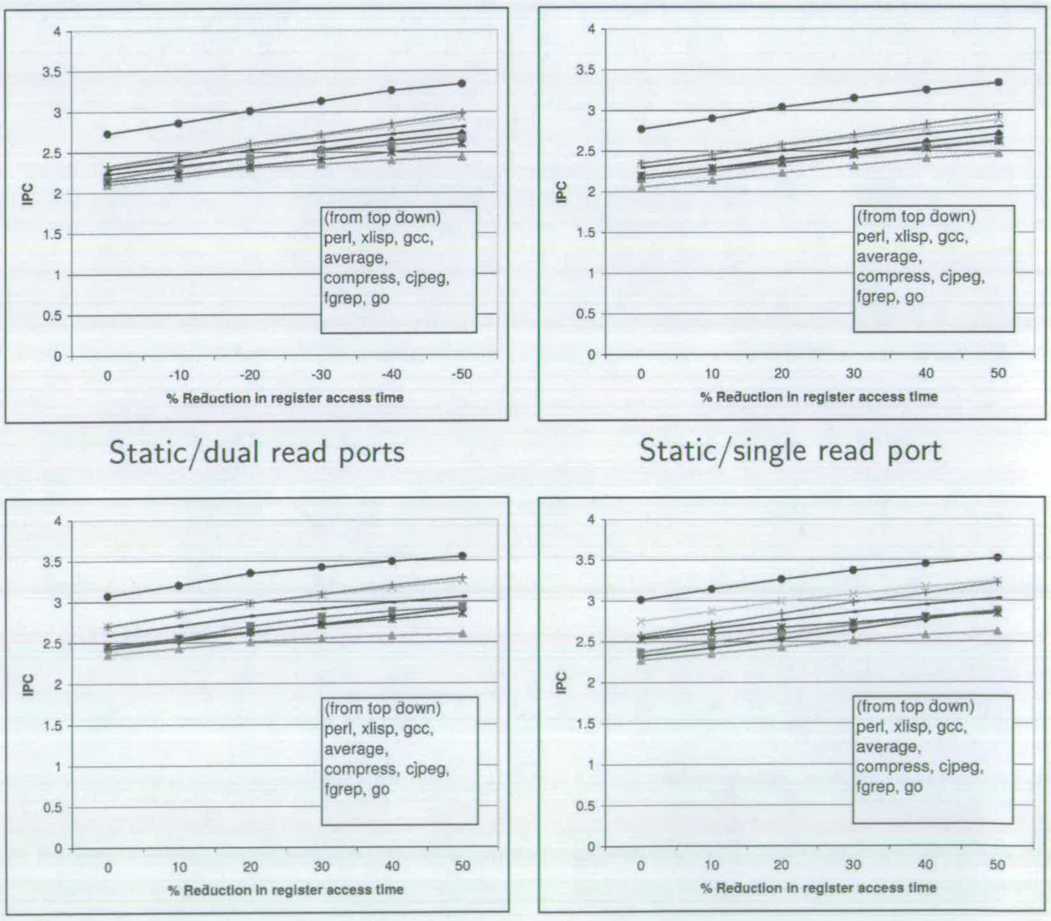


Figure 5.24: Percentage of forwarding operations cancelled. Results are shown for both static and dynamic cases, with single and dual register read ports per FU. (comp)



Static/dual read ports

Static/single read port

Dynamic/dual read ports

Dynamic/single read port

Figure 5.25: Impact of reducing register file access time. Results are shown for static and dynamic cases. Reduction ranges from 0-50% of original delay. (comp)

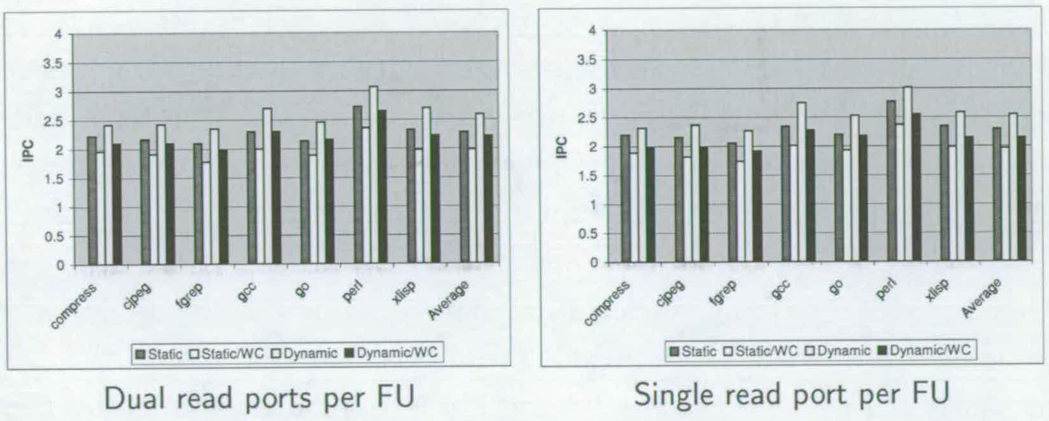


Figure 5.26: Impact of worst-case functional unit delays on static and dynamic compounding datapaths. Results are shown for both single and dual read port configurations. (comp)

5.3.4 Performance and complexity comparisons

The results in the previous sections show a disparity in performance between the models with full dynamic scheduling and data-forwarding and those without. This 35% difference in performance comes from the simpler models' inability to exploit the same levels of ILP, either through a lack of dynamic scheduling, or in the compounding case, due also to its inferior sequential performance as a result of a limited data-forwarding mechanism.

The performance of both these models will improve significantly with the introduction of appropriate compiler optimisations. This is particularly true in the case of the instruction compounding processor where its performance relies on careful selection and scheduling of compounds. One example of where the current random compounding algorithm often limits performance is in the case of loops. Here a poor selection of compounds can force each iteration of the loop to execute sequentially, even when there are no real loop carried dependencies forcing this behaviour. In the architectures with centralised dispatch buffers, where dispatch is far less restricted, the execution of many iterations of a loop may easily be overlapped or pipelined at run-time. To fully exploit the compounding architecture would require the development of an optimising compiler aware of the implications of creating particular compounds. This analysis would have to be performed at a high level in conjuction with global scheduling algorithms. The development of such a compiler was deemed to be out of the scope of this thesis.

From a complexity perspective, both the compounding and queue-based models offer a significant reduction in the hardware required to perform dynamic scheduling. Neither require destination registers to be broadcast to all waiting operands, as is the case in the architectures that exploit traditional dispatch buffer designs. The compounding and queue-based models also simplify the selection process, removing the need to arbitrate between a potentially large number of ready instructions. While dispatch buffers typically consume a small percentage of total chip area, when compared to caches and the rest of the datapath, their design is often non-trivial due to performance requirements [41]. The need to use larger transistors to minimise delays and the broadcasting of results also leads to significant power requirements when compared to other datapath components [56].

A good example of an asynchronous architecture's ability to exploit average-case performance to enable a reduction in complexity, is illustrated by the reduction in the number of register read ports required at each FU. This reduction in complexity has shown to be possible without a significant performance penalty.

A concise overview of the performance of each of the models presented is shown in Figure 5.27.

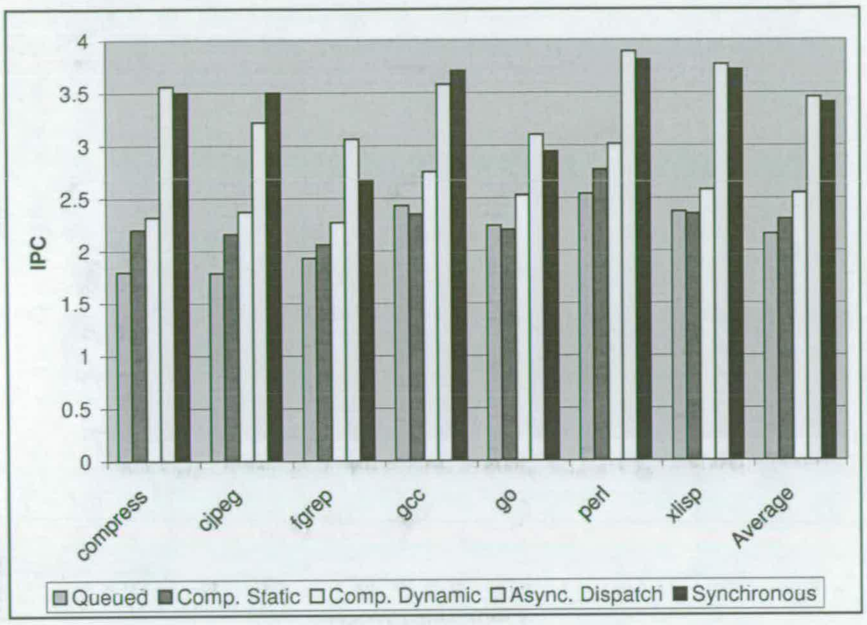


Figure 5.27: Performance comparison of best performing configurations for each processor model.

Chapter 6

Conclusions and Future Work

6.1 Summary

The adoption of an asynchronous design style offers many potential advantages in terms of power, performance, and design complexity. The extent to which these advantages may be exploited in a superscalar processor is largely dependent on its architecture. Design decisions made at a high-level heavily influence both overall performance and an architecture's ability to exploit a particular timing regime.

In Chapter 3, data-forwarding and dynamic scheduling are identified as both key features required to achieve high-performance and schemes whose implementation are traditionally dependent on the existence of global synchronisation. When it is considered that these mechanisms provide communication between instructions, it is unsurprising that their implementation is influenced by the choice of control paradigm. In the case of an asynchronous implementation, forcing such inter-instruction communication to take place simultaneously effectively synchronises the operation of the instruction execution pipelines. This type of pseudo-synchronous operation has a number of negative effects. Firstly, such synchronisation in an asynchronous system would represent a performance overhead, as it would most likely be exposed on the critical path of the control logic. Secondly, they would prevent the processor from exploiting actual datapath delays exposed by an asynchronous implementation; instead performance would tend towards the worst case. An approach which simply bases an asynchronous implementation on a synchronous architecture will most likely benefit from the retention of the clock signal. This ensures that the benefits of synchronous operation, such as predictability and determinism, can be fully exploited.

Both Chapters 3 and 4 detail a number of data-forwarding and dynamic scheduling schemes targeted at an asynchronous implementation. First to be examined is the design of an asynchronous dispatch buffer; here careful handling of the arbitration required at the buffer is necessary to maintain performance. In the design presented, concurrency is maximised by providing a data-dependent arbitration scheme where mutual-exclusion is only enforced when necessary. Two implementations of the selection logic were also explored.

The first of the data-forwarding schemes exploits information about the dispatch order of instructions at each functional unit. This makes a distributed implementation of the forwarding mechanism possible. The issue unit is used to maintain a table of those results which may be provided via a forwarding bus. A more general data-forwarding scheme is required to support the centralised dispatch buffer, as assumptions about the dispatch order of instructions can no longer be made. In the scheme presented, this requires that some forwarding requests are cancelled when it is determined that the data is not available for forwarding. A technique was devised as an extension to the dispatch buffer to reduce the number of forwarding requests which are subsequently cancelled. A counter associated with each operand in the dispatch buffer allows cases to be detected where forwarding will be unsuccessful; in such cases forwarding is aborted before the instruction is dispatched. The resulting reduction in the levels of forwarding request traffic aids the performance of those requests that are successful.

Chapter 4 introduced a compounded instruction processor. Here groups of dependent instructions are formed into compounds at compile-time to enable an alternative approach to be taken to both the design of the data-forwarding and dynamic scheduling mechanisms. Within instruction compounds, results are communicated via a data-forwarding mechanism, while between compounds communication is achieved through the register file. When forwarding is possible, information appended to the instruction producing the result allows it to make a forwarding request to a consumer (next instruction in the compound). In addition to providing a mechanism for forwarding data, this also forms part of the dynamic scheduling mechanism's wake-up process, indicating the availability of a particular operand. As a result, the instruction compounding architecture is able to provide a dynamic scheduling mechanism without the need to broadcast result tags. In addition, careful design of the dispatch logic also prevents the need to select from a large number of ready instructions.

The performance characteristics of each of the architectures described have been explored in the previous chapter. The asynchronous architecture with the highest performance, based around the dispatch buffer described in chapter 3, is shown to have broadly similar performance characteristics as the synchronous implementation. This is promising as it shows that an effective superscalar architecture can be devised without the support of global synchronisation. Performance results for the other, less complex, asynchronous models is shown to be lower than that of the synchronous model. A major consideration here is the lack of an optimising compiler which will benefit these architectures much more than those with more costly dynamic scheduling schemes. The use of a single register read port at each functional unit was also explored as a technique to reduce overall complexity. All the asynchronous models showed an ability to maintain overall performance after such a modification, even with the need to sequentialise a percentage of register reads. This type of complexity-effective optimisation is discussed further in the following section.

6.2 Future Work

Future work could be pursued in any of four major areas: architecture, verification, optimising compilers, and implementation.

In architectural terms, the adoption of an asynchronous design methodology may offer greater flexibility in applying techniques to reduce power consumption. One area in particular that may provide opportunities to exploit asynchronous operation is in the development of run-time reconfigurable components. Recent examples of such an approach have explored the design of run-time reconfigurable caches [13] and dispatch buffers [43]. While at present this work is limited to synchronous designs, with the aim of reducing power consumption, its application to asynchronous designs is an interesting one. The ability of an asynchronous architecture to exploit local performance gains made possible by exposing data-dependent delays, should also enable configuration-dependent delays to be exploited. The application of such an approach is also simplified in the asynchronous case by the absence of global timing requirements. Another way in which power may be reduced is by carefully partitioning the architecture to create regions operating at different frequencies [128]. Only those functions which require the most costly environment in terms of transistor size, threshold voltage and supply voltage would be integrated into the high-performance core. Other functions would be supported in cheaper helper engines. An asynchronous implementation may provide a number of advantages in such a scenario. For example, communication between regions operating at different speeds could easily be supported without the need to incur large synchronisation penalties. An asynchronous approach also offers a wider range of implementation possibilities, which may be particularly useful in minimising static power requirements. For example, simple asynchronous circuits may be designed that maintain average-case performance while trading a reduction in complexity for a higher worst-case performance. Trade-offs in terms of datapath width and frequency of operation are also simple to employ in the absence of global synchronisation.

A second area of future work is in the area of formal verification. One difficult challenge when developing asynchronous processors is to show that there is no possibility of deadlock occurring during the execution of a program. This problem becomes even more complex when correct execution is dependent on a combination of compiler-based and architectural features, for example in the case of the SCALP processor. Related work in [16] has explored the use of typing calculi to aid in reasoning about the correctness in such cases.

A limiting factor in the performance of both the queue-based and instruction compounding asynchronous processors is the lack of an optimising compiler. On going work described in [134], discusses the development of global scheduling algorithms for asynchronous processors. The architecture used to develop the scheduling algorithms is not very different from the queue-based architectures explored here. Incorporating the compiler into the current simulation environment could provide a fairer evaluation of the queue-based models. Extending this compiler infrastructure to support compound selection and scheduling could also be the starting point for exploring more fully the performance potential of instruction compounding architectures.

The implementation of any of the asynchronous superscalar architectures presented would also be interesting. For a fair comparison to be made with existing high-performance architectures similar design techniques would have to be employed; these include: full custom design, the use of low voltage swing buses, transistor size optimisations and the use of a modern process technology. In an academic environment implementing a complete processor design using this type of design style is unrealistic. A more realistic challenge may be in providing implementations for each of the dispatch buffer designs and the forwarding register logic in order for a more detailed comparison of their performance and power characteristics to be made.

6.3 Discussion

The final question is whether an asynchronous design style will ever be adopted for the design of a commercial superscalar processor. Arguments suggesting that this will never happen usually cite the existence of a large number of tools designed specifically for designing synchronous systems or the body of knowledge tailored specifically at synchronous implementations. For many less complex and lower performance systems this may be true, as circuits are usually synthesized from high-level descriptions before entering a highly automated tool flow. For the design of the highest-performance processors these comments are less appropriate as the whole design process is a fully-custom one, requiring in most cases for designs to be considered at the transistor level.

The most likely reason for adopting an asynchronous design style will be as a result of the eventually unmanageable level of complexity and correspondingly unrealistic design team size required to design synchronous processors. The growing complexity of the synchronous approach is a result of the reliance on accurately predicting delays for both logic and interconnect. This complexity is also likely to increase sharply with the predicted introduction of large numbers of independent clock domains. While many of these problems may be solved by a GALS (Globally-Asynchronous Locally-Synchronous) approach for lower performance systems, their application in high-performance designs seems less realistic. The range of on-chip delays will also increase both due to the need to optimise designs for low-power, but also due to a large range of on-chip communication delays. Results presented in [12], already suggest that there are significant advantages in adopting delay-insensitive signalling schemes for on-chip communication.

Synchronous designs have benefited for many years from the existence of a limited range of on-chip delays and relatively low clock frequencies. The existence of global synchronisation is ideal when the primary goal is the minimisation of delays incurred through computation. This is especially true when pipeline stages can be balanced to make effective use of each clock period. As feature sizes decrease and levels of integration increase, the problem shifts from a need to focus on minimising computation delays, to one of managing complex communication requirements while also minimising power dissipation. In such an environment it is believed that asynchrony offers the flexibility to better exploit the underlying implementation technology.

Large investments in CMOS technology and the advances predicted for the next 15 years, mean it will probably remain the dominant implementation technology for many years to come. If the problems of rising power consumption, increasing communication costs, and difficulty in predicting delays continue it is likely that a switch to an asynchronous design style will soon provide significant advantages.

6.4 Conclusions

The main contribution of this work has been the development of data-forwarding and dynamic scheduling mechanisms suitable for asynchronous superscalar processors. These mechanisms form the main architectural challenges in developing asynchronous architectures with the ability to exploit instruction-level parallelism. The solutions presented have been compared quantitatively through simulation. Results have shown that asynchronous architectures can be devised that provide similar performance to synchronous ones. In addition, a number of architectures have been explored that aim to provide simpler implementations. While the performance of these architectures is at present lower, performance will improve with the development of compiler support.

The adoption of an asynchronous control paradigm enables new and interesting trade-offs to be explored at the architectural level. It is hoped that asynchrony will enable novel computational structures to better exploit future advances in deep sub-micron technologies.

Appendix A

Published Papers

A.1 A Fully Asynchronous Superscalar Architecture

Title:	A Fully Asynchronous Superscalar Architecture
Authors:	D.K. Arvind and R.D. Mullins
Presented at:	International Conference on Parallel Architectures and Compilation Techniques (PACT)
Place:	Newport Beach, California
Date:	October 1999
Publisher:	IEEE Computer Society Press

A Fully Asynchronous Superscalar Architecture

D. K. Arvind and Robert D. Mullins
Division of Informatics
The University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ, Scotland.
dka,rdm@dcs.ed.ac.uk

Abstract

An asynchronous superscalar architecture is presented based on a novel architectural feature called instruction compounding. *This enables efficient dynamic scheduling and forwarding of data based on local information, while maintaining the advantages of asynchrony in terms of exploiting actual delays. Results are presented in which statically and dynamically compounded architectures are compared against an equivalent synchronous superscalar architecture.*

1. Introduction

The design of high clock frequency processors leads to considerable physical problems in distributing the clock signal, high power dissipation and poor electromagnetic (EM) interference characteristics. The asynchronous design approach has been proposed as a solution to these problems [8], although the potential of multiple issue asynchronous architectures has not yet been fully explored. This paper introduces a technique called instruction compounding which better enables the advantages of asynchrony to be exploited in a superscalar architecture.

2. Synchronous Superscalar Architecture

This section highlights some features of a typical synchronous superscalar pipeline (see Figure 1) with out-of-order instruction issue. The pipeline is capable of fetching and executing multiple instructions on each clock cycle, and is typically supported by branch prediction and speculative execution in order to maintain a high instruction bandwidth.

The *instruction-issue buffer* implements, in essence, a limited dataflow capability, in holding instructions while their operands are being generated, and allowing ready instructions to issue out-of-order. The buffer may issue multiple instructions in a clock cycle to a number of functional units which operate concurrently. The operation of the instruction issue buffer can be split into two phases: *wakeup* and *selection*. The wakeup logic matches results generated by the functional units to the operands in the issue buffer; the selection logic determines which of the ready instructions should be issued to free functional units. These architectures may issue dependent instructions in consecutive clock cycles by waking instructions in the same cycle as their final operand is being produced. A network of result buses and bypass logic is used to obtain the correct operand values on the subsequent clock cycle, which is commonly termed as data forwarding.

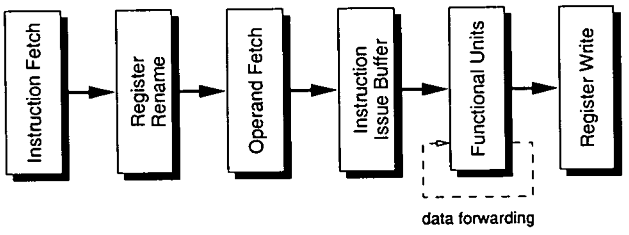


Figure 1. Synchronous Superscalar Pipeline

3. Asynchronous Superscalar Architecture

A number of synchronous implementations of the architectural features described previously already exist. Unfortunately imitating these designs within an asynchronous environment limits the extent to which the advantages of asynchrony may be exploited. To appreciate this statement we need to understand better the influence of the control paradigm on the architecture.

In synchronous architectures, the control mechanism has a rigid, periodic interaction with the datapath. Operations are initiated by the control unit and must complete within fixed multiples of clock cycles. This produces predictable and deterministic behaviour which may be exploited.

However the components of such a system must be designed to minimise the critical path to ensure a low clock period, even if this path is rarely taken. As a result, functional components lie idle for a proportion of the clock period, even though utilisation is high when measured in clock cycles. This is essentially a time-driven approach to the design of the interface between the control and the datapath. In contrast, one can implement an event-driven version of this interface using asynchronous circuits. This exposes actual delays within the datapath and results in components being active only when performing useful computations. A good asynchronous architecture is one which translates these local timing benefits to a better overall system performance. One way in which this may be achieved is by exploiting greater sub-instruction parallelism.

In synchronous implementations, both the instruction buffer and data forwarding mechanisms exploit global synchronisation. In the absence of a clock, a naive implementation would require a large number of local synchronisations - swamping any gains of exposing actual delays. We propose novel architectural ideas for efficiently realising dynamic scheduling and data forwarding in a fully asynchronous environment.

3.1. Novel ideas for instruction execution

In this section, we describe the design of an asynchronous superscalar processor, with emphasis on its out-of-order instruction execution and data-forwarding capabilities.

The basic pipeline, outlined in Figure 2, differs from the synchronous one described previously in the way that operands are obtained and instructions are scheduled. These operations are now distributed to execution units associated with each functional unit.

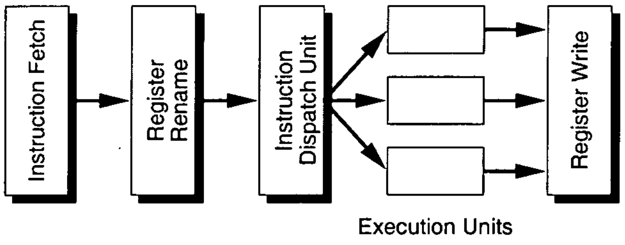


Figure 2. Asynchronous Superscalar Pipeline

The efficient operation of these execution units relies on information being obtained from the compiler in identifying possible candidates for data forwarding. The mechanism used to provide such information is called *instruction compounding* [1]. Instruction compounds provide additional information regarding the dependences between instructions. This information is used to reduce synchronisations between functional units when required to perform data forwarding and dynamic scheduling. A compound can be simply defined as a group of dependent instructions. A more precise definition with respect to the architecture is given below.

A basic block is partitioned into compounds by grouping adjacent dependent instructions. The only constraint in the selection of compounds is that the resulting graph of compounds must be a DAG. Within the compounding architecture results may only be forwarded between successive instructions within a compound. The example in Figure 5 illustrates a possible compounding for the code fragment. Instructions 2,3 and 4 are grouped together to form a compound, each instruction within the compound must be scheduled consecutively as shown. This allows membership of a particular compound to be indicated by a single *compounding bit* for each instruction. When the bit is set the instruction and the following instruction are both part of the same compound.

The architecture of an execution unit is shown in Figure 3. Instructions are issued out-of-order (and asynchronously) from the instruction buffer as soon as it is safe to do so. This is indicated by forwarding requests from other execution units, or the setting of future bits as other instructions issue. Once an instruction is ready and has successfully arbitrated for issue then its operands are obtained and its result is generated. Concurrently, a pipeline determines whether the result is to be forwarded, both finally converge in the forwarding unit from where data is actually forwarded. A more detailed description of the operation of these units follows.

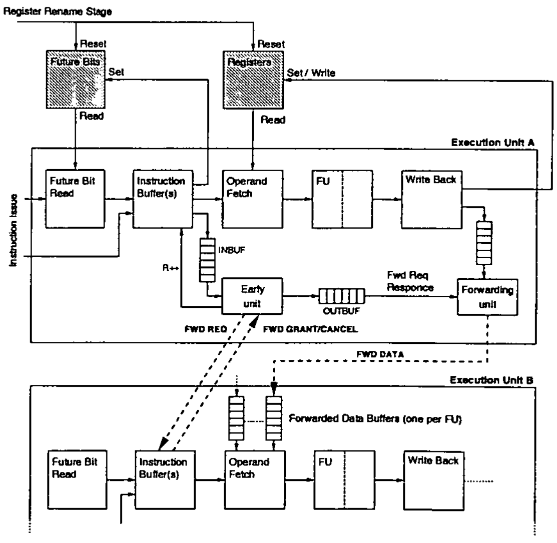


Figure 3. Execution Unit

Execution units receive results either via the shared register bank or directly from other units over the forwarding network.

In the absence of global synchronisation, communication via the register bank is implemented through the use of a register locking mechanism [6]. A status bit is attached to each physical register to indicate when its contents is valid. In addition, a *future bit* is associated with each register to indicate whether the instruction which will write to the register has been issued. Future bits guarantee the availability of results and are used to determine when an instruction may issue safely, without resulting in a deadlock. Both register status and future bits are reset during register renaming when a new physical register is mapped.

Once an instruction is dispatched to an execution unit, each of its operands which cannot be forwarded must read its register future bit. This is achieved by queuing each read operation in one of two read queues. After a future bit is read, the status of the corresponding operand in the instruction buffer is updated. This write is made using an instruction buffer write port. This operation is illustrated in Figure 4.

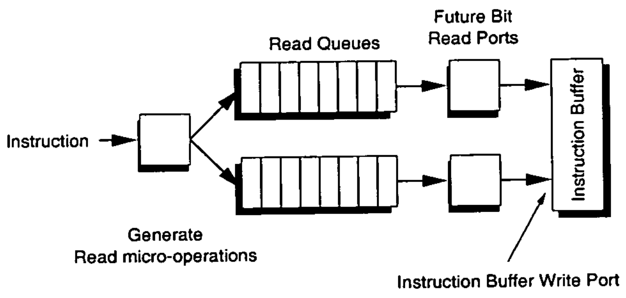


Figure 4. Future Bit Read Operation

The instruction buffer dispatches instructions out-of-order depending on the status of their operands. The operand status is updated via the instruction buffer write ports, either by successfully reading a future bit (as described before), or by receiving a forwarding request (to be described). These operations are equivalent to an instruction *wakeup* phase in a synchronous pipeline. Each write port is associated with an issue port. If a write wakes an instruction in the buffer, then the write port is blocked until the issue request is granted. This limits the number of arbitrating instructions to the number of write ports, which is desirable in asynchronous architectures (due to the delay of multi-way arbiters). Each write is made directly to a particular buffer entry - this is possible as both forwarding requests and future bit reads are tagged with the instruction's buffer entry.

Each entry in the buffer contains information about a particular instruction's operands, their status and forwarding bits, the operation to be performed at the functional unit, the instruction's compounding bit and the location of the next instruction in the compound. The forwarding bits associated with each operand indicate whether the result will be forwarded, or fetched from the register bank. These bits are initially set in the instruction dispatch unit.

Once an instruction has issued, it proceeds to the operand fetch stage, and should its compounding bit be set (it forwards its result), then it is also sent to the early unit input buffer. The future bit associated with its destination register will be set to indicate that the result is being generated.

The early unit queries the next instruction in a compound to determine if forwarding is possible. This query or *forwarding request* is also used to update the status of the operand. A detailed description of this operation is given below.

- The early unit receives each instruction which is a member of an instruction compound (bar the final instruction) and makes a forwarding request to the next instruction in the compound. The location (execution unit and buffer entry) of this instruction is obtained within the instruction dispatch unit.
- The forwarding request must arbitrate for access to the instruction buffer. Forwarding requests are then queued before they access a particular entry via a write port.
- When a forwarding request is made to a particular instruction in the buffer, then one of two situations will arise:
  - The status of all other operands has been updated through future bit reads. In this case, data forwarding is possible and the instruction may issue.
  - Future bit reads are pending for one or more operands. In this case it is not possible to issue the instruction and data forwarding must be cancelled. The operand which would have been forwarded is now obtained from the register bank, and its forwarding bit is reset to reflect this.
- The early unit will receive either an acknowledgement or cancellation signal. This information is used to determine whether or not to forward the data at the forwarding unit.

The order in which results are consumed from a particular execution unit must be guaranteed to be the same as the order in which they are sent. This is only possible by cancelling the forwarding of certain results. The alternative of issuing an instruction whenever it receives a forwarding request is not possible without introducing the possibility of deadlock.

Another potential deadlock condition involving the early unit is controlled by the release of instructions from the instruction buffer. Instructions are only released when there is no possibility of filling the early unit input buffer. The R++

signal in Figure 3 is used to maintain a count within the instruction buffer and implement such a mechanism. If the queue was to block instruction issue, then deadlock could occur.

Operand fetch obtains register and forwarded result data. Forwarded results are received into an individual queue for each sender. This is necessary as the order in which forwarded results are sent is only guaranteed with respect to a single execution unit. In both the cases of register operands and forwarded results, operand fetch will stall until the data is available.

3.2. A Simple Example

In this section we illustrate the operation of the datapath through a simple example (see Figure 5) of forwarding and dynamic scheduling.

(1) r5=mem[r2+4]
(2) r1=mem[r2]
(3) r2=r1*321
(4) r3=r2+r5

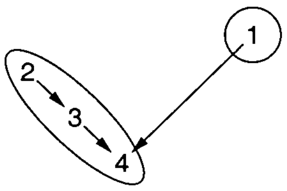


Figure 5. Example Compounds

Instructions 2,3 and 4 are compounded, while instruction 1 remains a singleton compound. Alternatively, compounds (2,3) and (1,4) could have been created. For simplicity, we assume in this example that the logical and physical registers used for each instruction have the same identifiers.

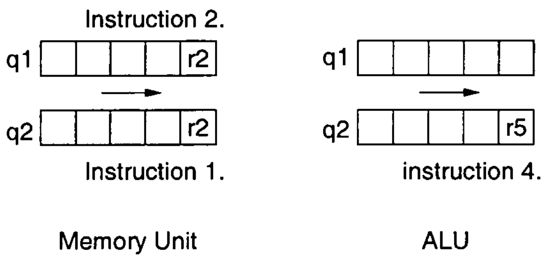


Figure 6. Future bit Read Queues

The following description shows how instructions are issued and obtain forwarded data.

- All instructions are dispatched to their respective execution units. In this case, a single memory unit (for instructions 1 and 2) and ALU (for instructions 3 and 4) are present.

- Future bit read operations are queued for all register operands (see Figure 6). Communication between instructions 2,3 and 4 are handled by forwarding operations and do not require future bit reads.

- We now concentrate on the execution of instructions 2 and 3. Instruction 3 requires no future bit read and only awaits a forwarding request from instruction 2. Instruction 2 issues after its operand's status bit is updated upon completion of the future bit read for register 2 (from q1 at the memory unit).

- The instruction proceeds to both the operand fetch stage and the early unit. The latter makes a forwarding request to instruction 3. Causing it to generate a forwarding acknowledge signal and to issue. Forwarding cannot be cancelled in this case as the instruction has no register operands.

- Once the result for instruction 2 is generated, the *forwarding unit* will receive both a result and forward request response - in this case an acknowledgement. The result will then be sent to the ALU's memory unit result queue, where instruction 3 will obtain the result during its operand fetch stage.

3.3. Dynamic Compounding

In the architecture presented so far instruction compounds are identified at compile-time. An alternative approach is to construct the compounds dynamically as instructions are read. This section describes an implementation of dynamic compounding, which extends compounding beyond basic block boundaries.

The implementation is based on a table being maintained within the register renaming, or issue stages of the datapath. An entry exists for each physical register and contains the following information:

- A *forward bit* to indicate that this result is to be forwarded. A destination in the form of a functional unit and instruction buffer entry is also present if the compounded bit is set.

- An *executed flag*, which is set once the instruction generating the result for this entry's register has queried the table.

An entry in the forwarding table is cleared when an instruction obtains its physical register destination. A subsequent read of this register may then be forwarded. This requires the compounded bit to be set in the table and the location of the instruction requiring the result to be recorded. A result may only be forwarded once, as in the static case, and only while the *executed flag* is clear. This flag is set when the instruction producing the result queries the table to see if the result is to be forwarded. This query takes place in an extra stage prior to the early unit. The details of the implementation have been omitted, as it is only used to explore the limits of compounding in this context.

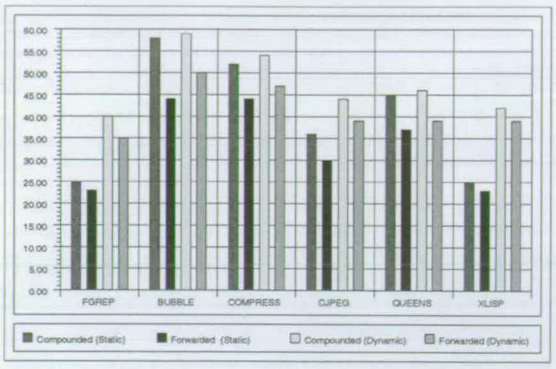


Figure 8. Percentage of operands compounded and results actually forwarded

are currently investigating a combined SW/HW approach to support forwarding across basic blocks.

Techniques also exist for exposing greater fine grained parallelism. For example, higher utilisation of the register read ports may be possible if the operand fetch stage is redesigned to permit each port to be accessed independently. This would allow both ports to be used if two instructions only require a single register fetch each. The overhead involved in implementing such aggressive techniques to expose further parallelism is current being evaluated.

5. Related Work

The effect of asynchrony on processor architecture has been explored in earlier work [3, 2], which introduced the notion of a fine-grained network of asynchronous agents called a *micronet*. Although this work was limited to scalar architectures many of the ideas and techniques for distributing control have been applied here.

Notable asynchronous processor implementations include an asynchronous MIPS R3000 [5] processor and the Amulet 2 [7], an asynchronous implementation of the ARM processor. Each makes some attempt to implement data forwarding, such as register bypassing in the case of the R3000 at the register bank, and by implementing last use registers in the Amulet 2 processor. A result forwarding mechanism designed for inclusion in the latest Amulet processor is presented in [4]. Here a small parallel FIFO is used to forward results between instructions currently in the pipeline. Each of these techniques have been developed for use within a scalar processor and their application to dynamically scheduled superscalar machines is limited. One reason for this is the large number of outstanding instructions and possible forwarding situations.

6. Conclusions

We have presented a novel architecture for exploiting asynchrony in superscalar architectures. To our knowledge this is the first detailed study into the performance advantages of an asynchronous multiple issue architecture.

We achieve better performance by two means: reducing run-time synchronisation and by exploiting fine-grained parallelism. Two techniques are used to achieve these aims. Firstly, instruction compounding reduces run-time synchronisations by generating forwarding information at compile time. Secondly, the early unit and future bits expose additional parallelism by allowing events to occur as early as possible while avoiding deadlock.

By understanding the interplay between compilers and architectures we aim to realise fully the performance potential of asynchronous multiple issue architectures.

References

[1] D. K. Arvind and R. D. Mullins. Instruction compounding. In *Proceedings of the 1st UK Asynchronous Forum*, Edinburgh, Scotland, Dec. 1996.

[2] D. K. Arvind, R. D. Mullins, and V. E. F. Rebello. Micronets: A model for decentralising control in asynchronous processor architectures. In *Asynchronous Design Methodologies*, pages 190–199. IEEE Computer Society Press, May 1995.

[3] D. K. Arvind and V. E. F. Rebello. Instruction-level parallelism in asynchronous processor architectures. In M. Moonen and F. Catthoor, editors, *Proc. of the Third Int. Workshop on Algorithms and Parallel VLSI Architectures*, pages 203–215. Elsevier Science Publishers, Aug. 1994.

[4] D. A. Gilbert and J. D. Garside. A result forwarding mechanism for asynchronous pipelined systems. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 2–11. IEEE Computer Society Press, Apr. 1997.

[5] A. J. Martin, A. Lines, R. Manohar, M. Nystroem, P. Penzes, R. Southworth, and U. Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181, Sept. 1997.

[6] N. C. Paver, P. Day, S. B. Furber, J. D. Garside, and J. V. Woods. Register locking in an asynchronous microprocessor. In *Proc. International Conf. Computer Design (ICCD)*, pages 351–355. IEEE Computer Society Press, Oct. 1992.

[7] S.B.Furber, J.D.Garside, S.Temple, J.Liu, P.Day, and N.C.Paver. AMULET2e: An Asynchronous Embedded Controller. In *Async '97*, pages 290–299. IEEE Computer Society Press, Apr. 1997.

[8] C. H. K. van Berkel, M. B. Josephs, and S. M. Nowick. Scanning the technology: Applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223–233, Feb. 1999.

4. Evaluation

We compare asynchronous architectures operating with statically and dynamically generated compounds to a synchronous superscalar machine. We also include results for a queue-based asynchronous architecture, which offers limited dynamic scheduling but lacks data forwarding.

All functional units share the same architectural parameters, as described (see Table 1). The delays used within the asynchronous architectures, as listed in Table 2, these are expressed as a percentage of the synchronous architecture's clock period.

Parameter	Number
No. of instrs. fetched per memory cycle	4
Complex ALU (ALU, logic, shift., mult.)	1
ALU (ALU, logic)	2
Memory Unit	1
Logical Registers	32
Physical Registers	64
Instruction buffers per Functional Unit	16

Table 1. Architectural Parameters

Component	Delay (% Cycle time)
Memory Access	100
Register Access	100
Future Bit read/write	60
Instruction Buffer Issue	50
Instruction Buffer Write	30
FU to FU communication	
Request (requires arbitration)	45
Acknowledge	30
Data	30
Instruction Delays	
ALU (add/shift)	50
Logical	20
Set/Move/Clear	0
Load/Store	100

Table 2. Asynchronous Component Delays

The following list gives additional implementation details specific to each model.

- The queue-based asynchronous architecture simply issues instructions to execution units consisting of an instruction queue, operand fetch stage and functional unit pipeline.

- The synchronous machine's instruction buffer is distributed amongst the functional units.

- In the case of the compounding architecture, each instruction buffer is split in two. One buffer is used to hold instructions which may receive forwarded data, and the other for those which will not. Two future bit read queues and ports are shared between each buffer, within each execution unit. Read operations were assigned in a round-robin fashion to the queues.

- The forwarding table (for dynamic compounding) incurs no delay due to reading, writing or arbitration. In this case, dynamic compounding is simply used to explore the possible advantages of extending compiler based compounding beyond basic blocks.

Results where obtained using a trace-driven, event-based simulator. The benchmarks used are *cjpeg* (spec95), *bubble sort, queens, compress*(spec92), *xlisp* (spec92) and *fgrep*. Instruction compounds were selected using a greedy graph partitioning algorithm with a maximum compound length of 10. No optimisations were performed on the schedule of compounded instructions, or for the queue-based asynchronous model.

Results showing the IPC (a cycle is defined in terms of a memory access operation for all the models) for each processor model are presented in Figure 7. Perfect branch prediction, memory disambiguation and instruction fetch bandwidth are assumed.

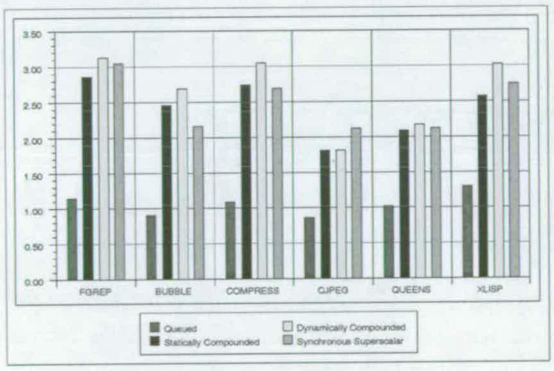


Figure 7. IPC for different processor models

The percentage of operands which were compounded either statically or dynamically and the actual percentage of operands obtained via forwarding are given in Figure 8. These differ due to the need to cancel some forwarding operations to avoid deadlock at run-time.

It can be seen from the results that the synchronous processor only outperforms the dynamically compounded model in one case (*cjpeg*). Static compounding performs worse than dynamic in all cases, only outperforming the synchronous model in the case of *bubble sort* and *compress*.

These preliminary results are encouraging, and they will improve with compiler optimised static compounding. We

A.2 Instruction Issue and Data Forwarding Mechanisms for Asynchronous Superscalar Processors

Title:	Instruction Issue and Data Forwarding Mechanisms for Asynchronous Superscalar Processors
Authors:	D.K. Arvind and R.D. Mullins
Presented at:	Workshop on Complexity-Effective Design (WCED'00), held in conjuction with ISCA-27
Place:	Vancouver, B.C.
Date:	June 2000

Instruction Issue and Data Forwarding Mechanisms for Asynchronous Superscalar Processors

D.K. Arvind and R.D. Mullins[†]
Division of Informatics
The University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ, Scotland.
dka@dcs.ed.ac.uk or Robert.Mullins@cl.cam.ac.uk

Abstract

An asynchronous design methodology offers potential advantages for architectures implemented in deep sub-micron technologies, such as low power dissipation and good electro-magnetic compatibility. This paper explores the impact of such a methodology on the architecture of superscalar processors. We examine in particular out-of-order instruction issue and data forwarding in the absence of global synchronisation. Three schemes are presented, and the performances of the resulting asynchronous superscalar architectures are compared to an equivalent synchronous one for a set of well-known benchmarks.

1. Introduction

The design of high-performance processors is becoming increasingly complex. This is due to a combination of factors such as their sheer size, architectural complexity, and the inherent difficulties of designing in deep sub-micron technologies [6], such as interconnect delays and limits on power dissipation and peak currents. The generation of global clock signals in the gigahertz frequency ranges is challenging, which is further compounded by requirements to gate clocks and support multiple on-chip clock domains.

An alternative and radical approach is to remove the clock altogether, and adopt an asynchronous design style [17]. At the architectural-level, there is no longer a need to distribute the clock globally, and the control structure can potentially exploit a range of computational and communication delays, without being limited to the worst-case timing behaviour. At the circuit-level, asynchrony attempts to minimise superfluous switching activity, which naturally benefits power consumption. The use of delay-insensitive interfaces and circuits leads to a modular approach to processor design which is free of complex timing requirements. This will become increasingly important as verification and design re-use become critical factors in design methodologies which are qualified by time-to-market considerations.

An asynchronous approach often requires novel design solutions to problems which have traditionally exploited global synchronisation. Naive solutions which operate in a pseudo-synchronous manner are often uninteresting, as they expose the full overhead of asynchronous handshaking and completion detection. A case-in-point is the handling of data dependencies in synchronous superscalar processors, viz. out-of-order instruction issue and data-forwarding, which rely on global synchronisation, and pose challenges in developing efficient asynchronous solutions.

[†]Robert Mullins is now with the Rainbow Group, University of Cambridge Computer Laboratory.

The following section provides a brief overview of the core operations in a typical synchronous superscalar processor. Section 3 develops the idea of an asynchronous version of the central instruction window which attempts to minimise synchronisations which degrade performance. Section 4 discusses a queued asynchronous architecture and develops data forwarding mechanisms for both this and the architecture described in Section 3. An alternative approach to dynamic scheduling and data forwarding which relies on a compile-time analysis of possible data forwarding opportunities, called *instruction compounding*, is described in Section 5. Finally, simulation results are presented comparing the performances of synchronous and the asynchronous architectures.

2 Superscalar Processors

Figure 1 illustrates a generic superscalar processor. The pipeline is capable of fetching and executing multiple instructions in each clock cycle, and is typically supported by branch prediction and speculative execution in order to maintain a high instruction bandwidth.

We first describe the operations which take place during instruction issue and data-forwarding, which will clarify the descriptions in the later sections. (See [13] for more details about the stages in such a pipeline) In the rest of the paper, we assume the presence of register renaming, and references to registers always imply physical ones, unless otherwise stated.

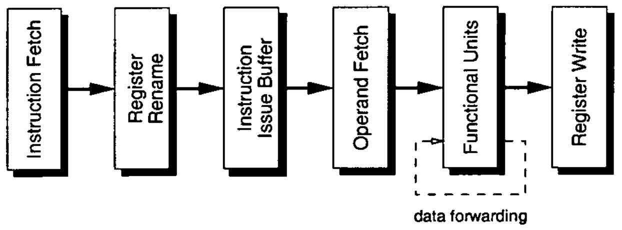


Figure 1. Synchronous Superscalar Pipeline

The central instruction window (CIW) or instruction issue buffer provides a limited dataflow capability, which allows instructions to be issued out of order, when both their operands and the appropriate execution resources are available. The basic operations performed within the window are listed below:

- **Write** New instructions are loaded into the window.
- **Wakeup** The status of the instruction's operands are updated, based on the lifetimes of issued instructions. It is useful to further decompose this phase into:
  - **Initialise** The operand's status is set on entering the window.
  - **Update** When instructions dependent on a particular result are issued, then all instructions which use the result are found and their status updated.
- **Selection** The ready instructions request to be issued. Instructions are selected for the available functional units.
- **Issue** A ready instruction is removed from the window and sent to the appropriate execution pipeline.

A pipelined processor includes a data-forwarding mechanism to permit dependent instructions to be executed in consecutive cycles. This mechanism enables data to be sent directly between functional units, bypassing the register file.

3 Transformation to an Asynchronous Design with CIW

The first asynchronous design is a gentle transformation of a typical synchronous central instruction window. We impose the following minimum requirement to avoid pseudo-synchronous operations: (1) we will not impose a strict order on the basic operations, as described in the previous section, which implement the dynamic scheduling mechanism, and (2) the operation of the functional units should not be synchronised by the scheduling mechanism. It should be possible to, issue to, and, handle results from, each FU independently.

Figure 2 shows a block diagram of the pipeline of such a design. An instruction in the CIW is ready for issue when all other instructions which produce its operands have been issued. An instructions initially obtains the current status of its operands (the *initialise* operation) from the register scoreboard (or more precisely, the status of the instructions which will produce its operands). The scoreboard operates in the following manner:

- An instruction at the rename stage (after renaming has taken place):
  1. sets the bit in the scoreboard corresponding to its destination register.
  2. reads the current status of the bits corresponding to its source operands.
  3. is written into the instruction window.
- An instructions leaving the instruction window:
  1. clears the bit corresponding to its destination register.

The status of instruction operands already in the instruction window are also updated as instructions leave the window. This is performed in the style of synchronous designs using content-addressable memory cells. The global CAM operation and the clearing of bits in the scoreboard together constitute the *update* phase.

In order to ensure that the status of the operands of each instruction is updated correctly, it is required that the *write* and *initialise* phases, and the *update* phase are not performed simultaneously. A single arbiter enforces mutual exclusion in this first design.

3.1 The Instruction Buffer

The CIW is itself implemented as a parallel FIFO [20, 19]. Reads and writes to each entry are controlled using head and tail pointers,

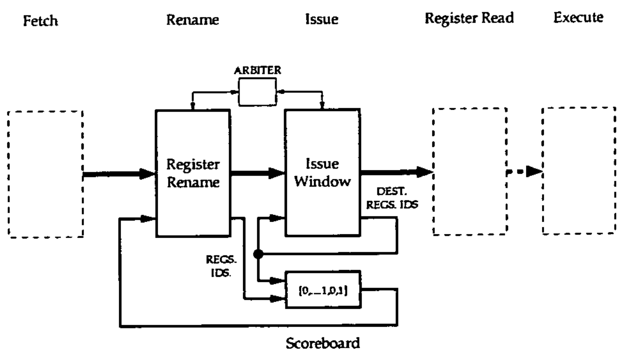


Figure 2. Asynchronous Central Instruction Window

which effectively implements a circular buffer. The fact that entries in this buffer do not move between memory elements simplifies our implementation of the instruction window. Other asynchronous FIFO implementations such as micropipelines [16], although offering other advantages, are problematic in this respect.

3.2 The Update and Selection Processes

Instructions become ready for issue once the status bits associated with each of their operands have been reset, which may occur either initially when the scoreboard is read, or after subsequent update operations. At the point when an instruction is ready, it will arbitrate for issue by raising a ready signal from its entry in the instruction window.

In order to avoid the need for F, N-input arbiters (where F and N are the number of functional units and entries in the window, respectively) and to ensure that the selection of older, ready instructions can be prioritised, a static selection phase is created for each functional unit. Static inputs to the selection circuit are provided through the use of selection arbiters as shown in Figure 3.

Each entry in the window holds a single instruction and is capable of raising a ready signal for each functional unit. Once all the operands of a particular entry are available, then a request is made to raise the ready signal for the

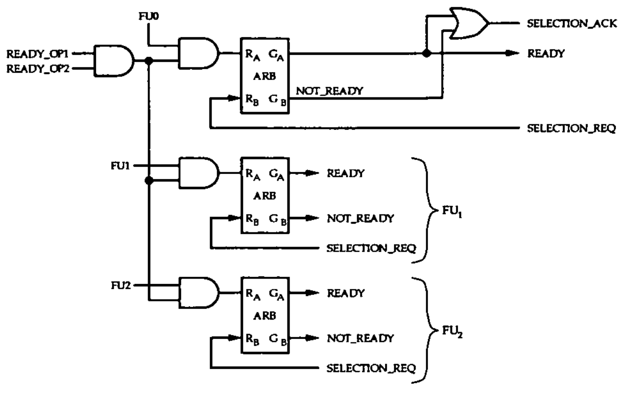


Figure 3. Selection Arbiters

target functional unit. In Figure 3, the signals *Ready_op1* and *Ready_op2* will be set if both operands are ready. This will result in a request to one of the selection arbiters corresponding to the target functional unit.

The selection and issue of instructions is outlined below. This process is performed concurrently for each functional unit.

- At least one of the ready signals is raised.
- The *selection_req* signal is asserted.
- All entries respond with a *selection_ack*. This is either a *ready* or *not_ready* signal (for the sake of clarity this is indicated for only the uppermost functional unit in the diagram). The inputs to the selection circuit are now guaranteed to be static and selection may take place.
- The selected instruction may be issued once the appropriate issue port is free (one port is associated with each execution pipeline).

The exact behaviour of the selection request signal is not specified here. A number of possibilities exist for controlling when the signal is to be asserted or deasserted. For example, we may wish to issue all ready instructions before deasserting the selection request signal, or deassert it after each issue in order maximise fairness. The point at which the selection request is raised is also undefined; one question is whether to make this dependent or independent of the state of the issue port. We do not elaborate further on these questions here as they depend on other architecture-specific parameters.

3.3 Reducing Synchronisation

The architecture as presented so far forces the write and initialise phases and update phase to be mutually exclusive. In practice, this is only required when each phase attempts to access the same entry in the scoreboard. Synchronisation may be reduced by providing an arbiter at each entry in the scoreboard, as shown in Figure 4.

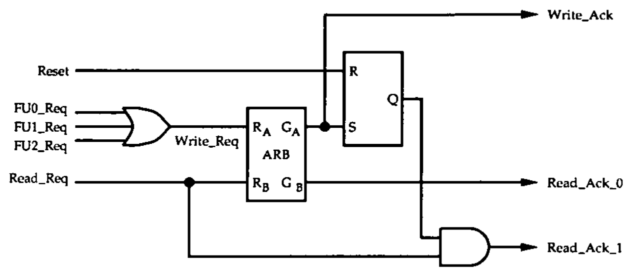


Figure 4. Arbitrated access to an individual scoreboard entry

Renaming guarantees that no two instructions with the same physical destination register may be simultaneously in flight. This ensures that no two functional units will ever attempt to access the same scoreboard entry at the same time. This enables a simple OR gate to generate the functional unit write request to the scoreboard entry. Access to the scoreboard is now arbitrated as before, but only enforces mutual exclusive operation, when both the rename stage and an update operation need to access the same entry.

Arbitration is only required within a particular entry, when it is possible for the entry to change its state from indicating a *not_ready* status to a *ready* one. No arbitration is required in the case when the entry is indicating a ready status (the RS flip-flop is set in the example).

Dependencies within a group of instructions which are fetched together may be determined without reference to the scoreboard during the rename process. This reduces the number of scoreboard reads and also enables the status of all the scoreboard entries associated with the destination registers of the instructions in the group to be reset simultaneously.

4 Data Forwarding in a Queued-based Architecture

Figure 5 shows an alternative asynchronous architecture with a limited form of dynamic scheduling. The monolithic instruction window has been replaced with a number of instruction queues. While this allows instructions at the head of each queue to be issued whenever dependencies permit, execution at a particular functional unit is always in order. The following section describes how such an architecture may be extended to support forwarding, and how a similar scheme may be used in general with the kind of central instruction window described previously.

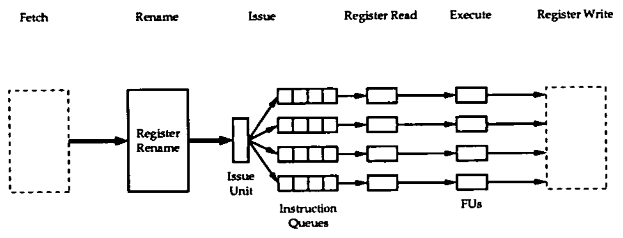


Figure 5. Queued Based Dynamic Scheduling

If it is possible to determine whether data can be obtained via forwarding at the instruction issue stage, then it would avoid the synchronisation necessary in traditional forwarding mechanisms, which effectively synchronises the operations of all the functional units. The HADES architecture [4] exploits a similar forwarding scheme, although in the absence of instruction queues or parallel register reads.

Recent results are stored at the output of each functional unit in a number of forwarding registers. The issue unit controls both the point at which the contents of these registers may be overwritten and when data is forwarded from these registers to other functional units. The issue unit maintains a record of the results which will be generated by the previous N instructions issued to each functional unit. If these results are required by subsequent instructions, then forwarding may be initiated by sending a forwarding request to the appropriate forwarding register. These requests together with commands to enable the contents of the forwarding registers to be overwritten, are queued in a forwarding register command buffer in each functional unit.

The existence of many forwarding sources introduces the problem that a number of outstanding forwarding requests may result in data being forwarded to a particular functional unit in a non-deterministic order. One solution to this problem is to provide a unique forwarded data input buffer, for each possible source, at each functional unit. We can, of course, guarantee that the data arriving from a particular functional unit will be in order, as forwarding requests are queued. The correct source is selected by the instruction during operand fetch, using information obtained during issue.

A block diagram illustrating a single execute pipeline incorporating data forwarding is shown in Figure 6.

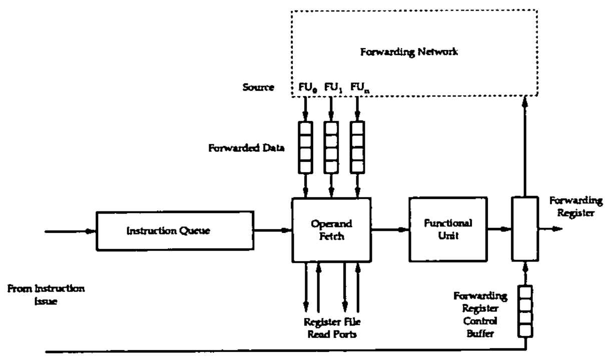


Figure 6. Execution Pipeline with Data Forwarding

A more general forwarding scheme is to enable the forwarding of results from the previous N instructions, irrespective of the functional unit they use. This requires that the individual forwarding registers be combined into a central forwarding queue. A similar structure, tailored for inclusion into the Amulet 3 is described in [9, 8]. The need to be able to forward results generated in a completely arbitrary order forces us to allocate entries in the forwarding queue during instruction decode. The queue must also be multi-ported for both reading (forwarding) and writing multiple results in parallel. This combination would probably preclude this type of design for the case when there are many functional units, each with its own input instruction buffer. Nevertheless, the queueing of forwarding requests and overwrite signals could be used in a manner similar to the distributed case described previously.

4.1 Data Forwarding and Central Instruction Windows

We now consider how data forwarding, as described in the previous sections, may be employed when instructions are issued in an arbitrary order (while respecting dependencies) from a central instruction window.

Consider the organisation shown in Figure 7, where just two functional units are represented. We now need to generate forwarding requests during the register read stage, as the order in which instructions will be executed is unknown prior to this. As in the previous description of a queue-based architecture, the source of data which may be forwarded is determined during issue. In the case of a central instruction window this information is appended to instructions in the window during the update phase. For example, if an instruction is issued to the memory unit, any operands requiring its result will record the memory unit as the source of data. Instruction operands which are initialised to be ready at the time the scoreboard is read will not generate forwarding requests. In reality such data will often already be available from the register file.

1. Instructions are issued to the appropriate operand fetch stage.

2. Forwarding requests are made, if necessary, for each operand. Operands which cannot be obtained via forwarding are fetching from the register file.

3. An overwrite signal is sent to a forwarding register at the output of the instruction's functional unit. This invalidates the contents of one of the forwarding registers in preparation for a new result to be latched.

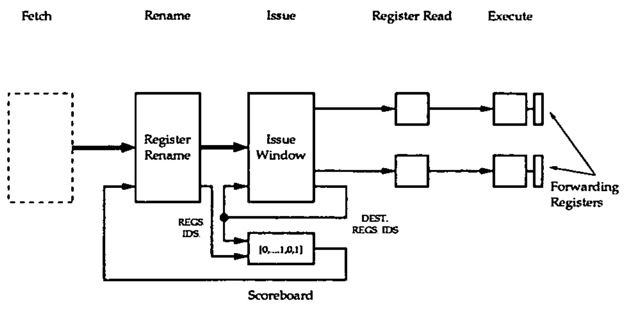


Figure 7. Asynchronous CIW Forwarding Problem

At this point it is clear that concurrent forwarding requests may be made from a number of operand fetch stages to a single forwarding register control buffer. One problem is that overwrite signals, invalidating the current contents of a forwarding register, may also be sent concurrently with forwarding requests for the current data.

This is solved by, firstly, arbitrating access to each forwarding register control buffer. Secondly, forwarding requests can no longer be guaranteed to be always successful. Forwarding requests may now occur too early - prior to the receipt of an overwrite signal preparing for the receipt of a particular result, or too late - after subsequent overwrite signals have effectively removed the data. To avoid incorrect data being forwarded, forwarding requests must now include the register identifier of the required data. A comparison in the forwarding register now determines if forwarding is possible, or if the forwarded operation must be cancelled, thereby forcing the value to be obtained via the register bank.

4.1.1 Source Counters

The cancellation of a large number of forwarding requests may result in a significant performance overhead. In order to reduce the number of surplus forwarding requests a mechanism may be introduced to cancel forwarding operations while instructions are still in the CIW.

A counter, which we will call the *source counter*, is associated with each operand in the CIW. This is initialised at the point when an update operation causes an operand entry to record a forwarding register as a source of data. Subsequent update operations from the same source now cause the counter to be decremented. When the counter's value is zero the source of the data is reset to the register file, a forwarding request will no longer be made.

5 Instruction Compounding

In the schemes discussed previously, forwarding has been initiated by the consumer. An alternative approach is to combine forwarding and dynamic scheduling, by allowing the producer to initiate a forwarding operation. This requires that dependencies be identified explicitly prior to issue. One way in which this may be achieved is to identify chains of dependent instructions at compile time. The following section gives an overview of such an approach, called instruction compounding - a more detailed description may be found in [1].

Instruction dependencies are identified by creating a number of instruction compounds during compilation, defined as groups of dependent instructions. A basic block is partitioned into compounds by grouping dependent instructions, the only constraint being that the resulting graph of compounds must be a DAG. Results may only be forwarded between successive instructions within a compound. The example in Figure 8 illustrates a possible compounding for the code fragment. Instructions 2, 3, and 4 are grouped together to form a compound, each instruction within the compound must be scheduled consecutively as shown. This allows membership of a particular compound to be indicated by a single compounding bit for each instruction.

```
(1) r5=mem[r2+4]
(2) r1=mem[r2]
(3) r2=r1*321
(4) r3=r2+r5
```

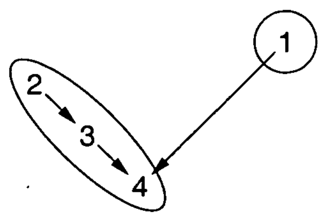


Figure 8. Example Compounding

The architecture consists of a number of independent instruction buffers and forwarded data result queues much like the queued-based architecture described in Section 4. A significant difference is that data-forwarding and instruction issue now exploit the explicit dependency information provided by compounding. An instruction, after it has been issued, is now able to make a request to forward its result to an instruction waiting in another instruction buffer. This partly implements both a forwarding and update operation without the need for the broadcasting of result identifiers. For those operands which may not be obtained via forwarding, initialisation of their status is provided by reference to a scoreboard much like the one described previously. Operations to read and update entries in the instruction buffers are queued upon issue.

Out-of-order instruction issue, as in the case described in the previous section, forces some forwarding operations to be cancelled. In the case of instruction compounding this is necessary when the instruction, on receipt of a forwarding request, cannot issue immediately. This is necessary to both avoid deadlock and guarantee that forwarded data is consumed in the correct order.

6 Results

Results were obtained from a trace-driven event-based simulator running well-known benchmarks such as *cjpeg*, *compress*, *fgrep*, *gcc*, *go*, *perl* and *xlisp*. Around 1 million instructions were simulated for each benchmark, the size of which is at present limited by the complexity of the asynchronous models. Instruction and memory reference traces were collected using QPT2 [3]. The Dinero cache simulator [10] is integrated into the simulator to provide information concerning cache misses. A 64k, 2-way set associative data cache and perfect instruction cache is used in all the simulations.

Instruction compounds were selected at random using a greedy graph partitioning algorithm with a maximum compound length set to 10. Note that no optimisations were performed on the schedule for either the compounded instructions, or the queue-based asynchronous model. Perfect branch prediction and instruction fetch bandwidth are assumed. Models which perform out-of-order instruction issue, excluding the queue-based architecture, are capable of executing memory instructions out of order with the aid of speculative memory disambiguation, although store-to-load forwarding is not implemented at present. Key architectural parameters are listed in Table 1. In the case of the queued and compounded models additional adders (equal to the number of memory units) are incorporated to generate memory addresses, in the architectures with central instruction windows ALUs are shared between address calculations and ALU instructions.

Parameter	Number
No. of instrs. fetched per mem. cycle	4
Complex ALU	1
ALU	2
Memory Units	1 or 2
Logical Registers	32
Physical Registers	80
Total No. of Instruction buffers	64

Table 1. Architectural Parameters

Results are included for three main asynchronous processor configurations, labelled as: *ACIW* (Asynchronous Central Instruction Window) and *ACIWc* (ACIW with source counters), *Queued* (the queue-based architecture described in Section 4) and *Comp* (Instruction Compounding). Where appropriate, results have been obtained for architectures containing 1 or 2 forwarding registers (f), and 1 or 2 register read ports (r), per functional unit.

Parameter	**Delay (% Cycle Time)**
Memory Access	100
Register File (8 read)	100
Register File (4 read)	78
Scoreboard Access	35
4-way arbitration	20
Instruction Delays	
ALU (add/shift)	50
Logical	20
Set/Move/Clear	0
Load/Store	100

Table 2. Delay Parameters

A selection of delay values, given as percentages of the synchronous clock period, are given in Figure 2. The smaller buffers or queues in the asynchronous models are assumed to have a throughput of 4 elements per cycle. Wakeup and selection in the case of the asynchronous CIW takes the equivalent of a full clock cycle.

Figures 9 and 11 summarise the IPC obtained for each model, each result being the geometric mean of the IPCs for each benchmark. Figures 10 and 12 record the percentage of operands which were obtained through data-forwarding for each model. Also shown, as a percentage of the total forwarding requests made, is the number of requests which were denied or cancelled.

In the case of the asynchronous CIW the number of forwarding requests which must be cancelled is significantly reduced with an additional forwarding register. This increases the window of opportunity for obtaining data through forwarding. Increasing the number of forwarding registers beyond two produces a reduction in performance for both CIW and queued models, for the delay models used.

In the case of both the asynchronous CIW and compounding models some scope exists for

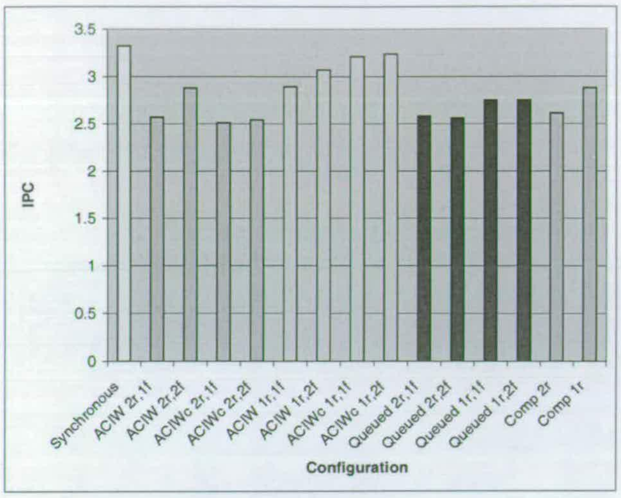


Figure 11. IPC for different processor models (two memory units).

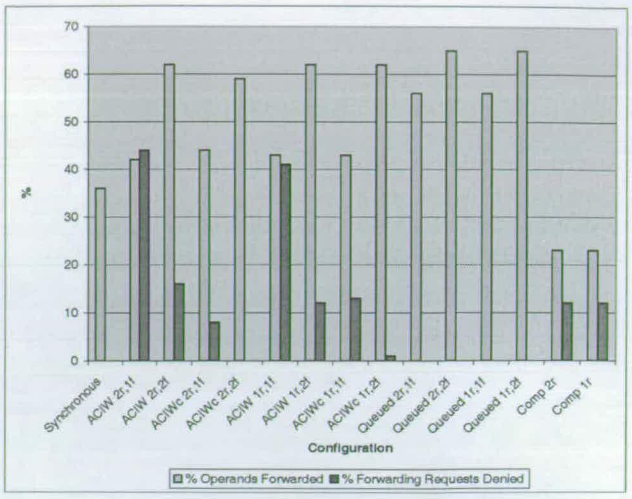


Figure 12. Percentage of operands forwarded and percentage of forwarding requests denied (two memory units).

the complex arbitration associated with the counterflow structure.

SCALP [5] was an attempt to develop a low-power superscalar processor, based on the ideas of explicit forwarding. The register bank is removed and instructions simply indicate the functional unit to which their result is to be sent. One problem of such an approach is that if results are required more than once, explicit duplicate operations must be performed. Problems also occur when the destination of a results cannot be determined at compile-time. A functional unit which is effectively a register bank is required to handle some communications.

Other implementations which have attempted to exploit asynchrony have used more traditional architectures [7, 14, 11]. These have provided promising power/performance results and have reported significant reductions in EMI.

8 Conclusions

This paper has presented three novel techniques for out-of-order instruction issue and data forwarding in the absence of global synchronisation. Their efficient implementation is important to the viability of asynchronous superscalar architectures. Results have been presented based on trace-driven simulations of detailed RTL models of the asynchronous architectures. Their performance is promising and there is scope for further improvement, in particular the compounding architecture could benefit from optimised compound selection and static scheduling.

Acknowledgements

The authors would like to acknowledge the support of the Institute for Computing Systems Architecture, Department of Informatics, University of Edinburgh. Robert Mullins would also like to acknowledge the Rainbow Group at the University of Cambridge Computer Laboratory for their kind support.

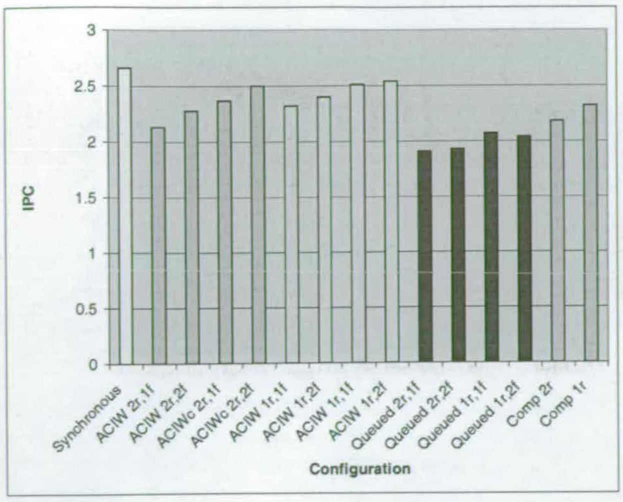


Figure 9. IPC for different processor models (single memory unit).

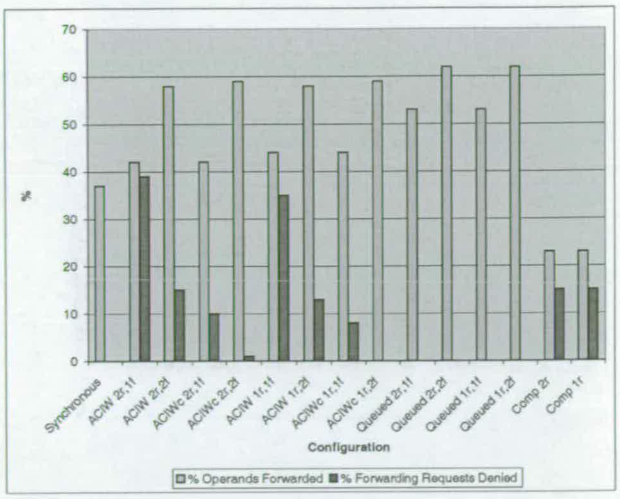


Figure 10. Percentage of operands forwarded and percentage of forwarding requests denied (single memory unit).

improving performance by reducing the number of these forwarding misses. Some attempts are already made to reduce misses in the current CIW design. For example, sources of forwarded data are reset when subsequent updates to the same instruction indicate a second operand will be obtained from the same FU. Results also indicate the potential of the counter-based scheme in reducing forwarding misses and increasing overall performance. Attempts to improve performance in the case of instruction compounding will centre on optimisations for compound selection and scheduling - which control both the forwarding of data and possible dynamic schedules.

Another factor influencing the performance of all of the asynchronous models is the reduction in the number of register file read ports. In all cases this improves performance due to a reduction in RF access time, which outweighs the penalty of serialising operand fetch in the relatively small number of cases where two or more registers must be fetched for a single instruction. The total number of operands forwarded differs significantly between the queued and compounding models; this is due to the need to restrict forwarding to between adjacent instructions and to within basic blocks. Compounding between basic blocks cannot be determined statically. The ability for all the models, including compounding, to outperform the queued model, even with a reduction in the levels of forwarding, is due to their more flexible dynamic scheduling schemes.

7 Related Work

The counterflow pipeline proposed in [15] consists of two pipelines flowing in opposite directions. Instructions in the instruction pipeline are able to inspect, and if necessary copy results from earlier instructions, as they flow past in the result pipeline. Unfortunately the orderings imposed upon data in the architecture may lead to local congestion and it has been suggested that from a purely performance viewpoint they do not compete with traditional architectures [2]. Attempts have also been made to exploit the counterflow pipeline structure to perform dynamic scheduling [18].

The rotary pipeline [12] again exploits a regular pipeline structure. Here results flow between a number of functional units organised into a ring. The availability of data on a particular bus is determined during issue and does not require

References

[1] D. K. Arvind and R. D. Mullins. A fully asynchronous superscalar processor. In *International Conference on Parallel Architectures and Compilation Techniques*, Newport Beach, California, October 1999.

[2] Bill Coates, Jo Ebergen, Jon Lexau, Scott Fairbanks, Ian Jones, Alex Ridgway, David Harris, and Ivan Sutherland. A counterflow pipeline experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 161–172, April 1999.

[3] Mark D.Hill, James R. Larus, Alvin R. Lebeck, Madhusudhan Talluri, and David A. Wood. WARTS Wisconsin Architectural Research Tool Set. http://www.cs.wisc.edu/ larus/warts.html.

[4] 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.

[5] Philip Brian Endecott. *SCALP: A Superscalar Asynchronous Low-Power Processor.* PhD thesis, University of Manchester, 1995.

[6] M. J. Flynn, P. Hung, and K. W. Rudd. Deep-submicron microprocessor design issues. *IEEE Micro*, pages 11–22, July 1999.

[7] Stephen B. Furber, James D. Garside, Peter Riocreux, Steven Temple, Paul Day, Jianwei Liu, and Nigel C. Paver. AMULET2e: An asynchronous embedded controller. *Proceedings of the IEEE*, 87(2):243–256, February 1999.

[8] J. D. Garside, S. B. Furber, and S.-H. Chung. AMULET3 revealed. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 51–59, April 1999.

[9] D. A. Gilbert and J. D. Garside. A result forwarding mechanism for asynchronous pipelined systems. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 2–11. IEEE Computer Society Press, April 1997.

[10] J. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.

[11] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nystroem, Paul Penzes, Robert Southworth, and Uri Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181, September 1997.

[12] S. Moore, P. Robinson, and S. Wilcox. Rotary pipeline processors. *IEE Proceedings, Computers and Digital Techniques*, 143(5):259–265, September 1996.

[13] Subbarao Palacharla, Norman P.Jouppi, and James E. Smith. Quantifying the complexity of superscalar processors. Technical report TR-96-1328, University of Wisconsin-Madison, November 1996.

[14] N. C. Paver, P. Day, C. Farnsworth, D. L. Jackson, W. A. Lien, and J. Liu. A low-power, low-noise configurable self-timed DSP. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 32–42, 1998.

[15] Robert F. Sproull, Ivan E. Sutherland, and Charles E. Molnar. The counterflow pipeline processor architecture. *IEEE Design & Test of Computers*, 11(3):48–59, Fall 1994.

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

[17] C. H. (Kees) van Berkel, Mark B. Josephs, and Steven M. Nowick. Scanning the technology: Applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223–233, February 1999.

[18] T. Werner and V.Akella. Counterflow pipeline-based dynamic instruction scheduling. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[19] Alexandre V. Yakovlev, Albert M. Koelmans, and Luciano Lavagno. High-level modeling and design of asynchronous interface logic. *IEEE Design & Test of Computers*, 12(1):32–40, Spring 1995.

[20] J. T. Yantchev, C. G. Huang, M. B. Josephs, and I. M. Nedelchev. Low-latency asynchronous FIFO buffers. In *Asynchronous Design Methodologies*, pages 24-31. IEEE Computer Society Press, May 1995.

Bibliography

[1] The National Technology Roadmap for Semiconductors. Tech. rep., Semiconductor Industry Assn., 1999. http://public.itrs.net/files/1999_SIA_Roadmap/Home.htm.

[2] Acosta, R. D., Kjelstrup, J., and Torng, H. C. An instruction issuing approach to enhancing performance in multiple functional unit processors. *IEEE Transactions on Computers C-35*, 9 (Sept. 1986), 815–828.

[3] Appleton, S., Morton, S., and Liebelt, M. High performance two-phase asynchronous pipelines. *IEICE Transactions on Information and Systems E80-D*, 3 (Mar. 1997), 287–295.

[4] Appleton, S. S., Morton, S. V., and Liebelt, M. J. The design of a fast asynchronous microprocessor. *IEEE Technical Committee on Computer Architecture Newsletter* (Oct. 1995).

[5] Arvind, D., and Sotelo-Salazar, S. Scheduling instructions with uncertain latencies in asynchronous architectures. In *EUROPAR'97* (Passau, Germany, Aug. 1997), Springer-Verlag.

[6] Arvind, D. K., and Hildingsson, K. Power tradeoffs in asynchronous interfaces. In *Asynchronous INTerfaces: tools, techniques, and implementations (AINT)* (Delft, The Netherlands, July 2000).

[7] Arvind, D. K., and Mullins, R. D. Instruction compounding. In *Proceedings of the 1st UK Asynchronous Forum* (Edinburgh, Scotland, Dec. 1996).

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

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

[10] Arvind, D. K., and Rebello, V. E. F. 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)* (Aizu Wakamatsu City, Japan, Mar. 1996), IEEE Computer Society Press, pp. 80–91.

[11] Austin, T. M., and Sohi, G. S. Zero-cycle loads: Microarchitecture support for reducing load latency. In *Proceedings of the 28th* (Ann Arbor, Michigan, November 29–December 1, 1995), IEEE Computer Society TC-MICRO and ACM SIGMICRO, pp. 82–92.

[12] Bainbridge, W. J., and Furber, S. B. Delay insensitive system-on-chip interconnect using 1-of-4 data encoding. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Mar. 2001).

[13] Balasubramonian, R., Albonesi, D., Buyuktosunoglu, A., and Dwarkadas, S. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-33)* (Dec. 2000).

[14] Bardsley, A., and Edwards, D. Compiling the language Balsa to delay-insensitive hardware. In *Hardware Description Languages and their Applications (CHDL)* (Apr. 1997), C. D. Kloos and E. Cerny, Eds., pp. 89–91.

[15] Benes, M., Nowick, S. M., and Wolfe, A. A fast asynchronous Huffman decoder for compressed-code embedded processors. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (1998), pp. 43–56.

[16] BERINGER, L. Typing assembly programs for asynchronous processors. In *Proceedings of the 9th UK Asynchronous Forum* (Cambridge, UK, Dec. 2000).

[17] BERKEL, K. V. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, vol. 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[18] BERKEL, K. V., BURGESS, R., KESSELS, J., PEETERS, A., RONCKEN, M., AND SCHALIJ, F. A fully-asynchronous low-power error corrector for the DCC player. In *International Solid State Circuits Conference* (Feb. 1994), pp. 88–89.

[19] BORKAR, S. Design challenges of technology scaling. *IEEE Micro* (July/August 1999), 23–29.

[20] BORMANN, D. S., AND CHEUNG, P. Y. Asynchronous wrapper for heterogeneous systems. In *Proc. International Conf. Computer Design (ICCD)* (Oct. 1997).

[21] BOSE, B., AND RAO, T. R. N. Theory of unidirectional error correcting/detecting codes. *IEEE Transactions on Computers C-31*, 6 (June 1982), 521–530.

[22] BRINGMAN, R., MAHLKE, S., HANK, R., GYLLENHAAL, J., AND HWU, W. Speculative execution exception recovery using writeback suppression. In *26th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-26)* (1993), pp. 214–223.

[23] BRUNVAND, E., AND SPROULL, R. F. Translating concurrent programs into delay-insensitive circuits. In *The Proceedings of the International Conference on Computer Aided Design (ICCAD-89)* (Nov. 1989), pp. 262–265.

[24] BUTLER, M., AND PATT, Y. N. An investigation of the performance of various dynamic scheduling techniques. In *25th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-25)* (1992), pp. 1–9.

[25] BUYUKTOSUNOGLU, A., SCHUSTER, S., BROOKS, D., BOSE, P., COOK, P., AND ALBONESI, D. An adaptive issue queue for reduced power at high performance. In *Proceedings of the Workshop on Power-Aware Computer*

Systems, held at the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (Nov. 2000).

[26] BYSTROV, A., AND YAKOVLEV, A. Ordered arbiters. *Electronic Letters 35*, 11 (1999), 877–879.

[27] CHANEY, T. J., ORNSTEIN, S. M., AND LITTLEFIELD, W. M. Beware the synchronizer. In *IEEE 6th International Computer Conference* (1972), pp. 317–319.

[28] CHAPIRO, D. M. *Globally-Asynchronous Locally-Synchronous Systems.* PhD thesis, Stanford University, Oct. 1984.

[29] COATES, B., EBERGEN, J., LEXAU, J., FAIRBANKS, S., JONES, I., RIDGWAY, A., HARRIS, D., AND SUTHERLAND, I. A counterflow pipeline experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Apr. 1999), pp. 161–172.

[30] CORPORAAL, H. *Microprocessor Architectures from VLIW to TTA.* John Wiley, 1998.

[31] CORTADELLA, J., KISHINEVSKY, M., A.KONDRATYEV, LAVAGNO, L., AND YAKOVLEV, A. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems E80-D*, 3 (Mar. 1997), 315–325.

[32] DEAN, M. E., DILL, D. L., AND HOROWITZ, M. Self-timed logic using current-sensing completion detection (CSCD). In *Proc. International Conf. Computer Design (ICCD)* (Oct. 1991), IEEE Computer Society Press, pp. 187–191.

[33] DEAN, M. E., DILL, D. L., AND HOROWITZ, M. Self-timed logic using current-sensing completion detection (CSCD). *Journal of VLSI Signal Processing 7*, 1/2 (Feb. 1994), 7–16.

[34] DESAI, U., TAM, S., KIM, R., ZHANG, J., AND RUSU, S. Itanium processor clock design. In *Proceedings of the international symposium on Physical design (ISPD)* (2000), pp. 94–98.

[35] D.HILL, M., AND EDLER, J. Dinero IV Trace-Driven Uniprocessor Cache Simulator. http://www.cs.wisc.edu/ markhill/DineroIV/.

[36] D.Hill, M., Larus, J. R., Lebeck, A. R., Talluri, M., and Wood, D. A. WARTS Wisconsin Architectural Research Tool Set. http://www.cs.wisc.edu/ larus/warts.html.

[37] Dwyer, H., and Torng, H. C. An out-of-order superscalar processor with speculative execution and fast precise interrupts. *SIGMICRO Newsletter* (1992), 272–281.

[38] Elston, C. J., Christianson, D. B., Findlay, P. A., and Steven, G. B. Hades - Towards the design of an asynchronous superscalar processor. In *The Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies* (London, UK, May 1995), M. B. Josephs, Ed., IEEE Computer Society Press, pp. 200–209.

[39] Endecott, P. B. *SCALP: A Superscalar Asynchronous Low-Power Processor.* PhD thesis, University of Manchester, 1995.

[40] Fair, H., and Bailey, D. Clocking design and analysis for a 600MHz Alpha microprocessor. In *IEEE International Solid-State Circuits Conference* (Feb. 1998).

[41] Fischer, T., and Leibholz, D. Design tradeoffs in stall-control circuits for 600Mhz instruction queues. In *International solid state circuits conference (ISSCC)* (San Francisco, CA, Feb. 1998), pp. 232–233.

[42] Floyd, B. A., Kim, K., and O, K. K. Wireless interconnection in a CMOS IC with integrated antennas. In *International Solid State Circuits Conference* (Feb. 2000).

[43] Folegnani, D., and González, A. Reducing power consumption of the issue logic. In *Proceedings of the Workshop on Complexity-Effective Design (WCED'00, held in conjuction with ISCA-27)* (Vancouver, B.C., June 2000).

[44] Furber, S. B., and Day, P. Four-phase micropipeline latch control circuits. *IEEE Transactions on VLSI Systems 4*, 2 (June 1996), 247–253.

[45] Furber, S. B., Day, P., Garside, J. D., Paver, N. C., and Woods, J. V. AMULET1: A micropipelined ARM. In *The Proceedings of IEEE CompCon'94* (Mar. 1994).

[46] Furber, S. B., Edwards, D. A., and Garside, J. D. AMULET3: a 100 MIPS asynchronous embedded processor. In *Proc. International Conf. Computer Design (ICCD)* (Sept. 2000).

[47] Furber, S. B., Garside, J. D., Riocreux, P., Temple, S., Day, P., Liu, J., and Paver, N. C. AMULET2e: An asynchronous embedded controller. *Proceedings of the IEEE 87*, 2 (Feb. 1999), 243–256.

[48] Furber, S. B., Garside, J. D., Temple, S., Liu, J., Day, P., and Paver, N. C. AMULET2e: An Asynchronous Embedded Controller. In *Async '97* (Apr. 1997), IEEE Computer Society Press, pp. 290–299.

[49] Gageldonk, H. v. The asynchronous move machine: Verification using CCS. Master's thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, Aug. 1994.

[50] Gageldonk, H. v., Baumann, D., van Berkel, K., Gloor, D., Peeters, A., and Stegmann, G. An asynchronous low-power 80c51 microcontroller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (1998), pp. 96–107.

[51] Garside, J. D. A CMOS VLSI implementation of an asynchronous ALU. In *Asynchronous Design Methodologies* (1993), S. Furber and M. Edwards, Eds., vol. A-28 of *IFIP Transactions*, Elsevier Science Publishers, pp. 181–207.

[52] Gilbert, D. A., and Garside, J. D. A result forwarding mechanism for asynchronous pipelined systems. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Apr. 1997), IEEE Computer Society Press, pp. 2–11.

[53] Ginosar, R., and Kol, R. Adaptive synchronization. In *Proc. International Conf. Computer Design (ICCD)* (Oct. 1998), pp. 188–189.

[54] Gonzalez, R., and Horowitz, M. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits* (Sept. 1996), 1277–1284.

[55] Gopalakrishnan, G. Developing micropipeline wavefront arbiters. *IEEE Design & Test of Computers 11*, 4 (Winter 1994), 55–64.

[56] GOWAN, M. K., BIRO, L. L., AND JACKSON, D. B. Power considerations in the design of the Alpha 21264 microprocessor. In *Proceedings of the 1998 Conference on Design Automation (DAC-98)* (Los Alamitos, CA, June 15–19 1998), ACM/IEEE, pp. 726–731.

[57] GRASS, E., MORLING, R. C. S., AND KALE, I. Activity monitoring completion detection (AMCD): A new single rail approach to achieve self-timing. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Mar. 1996), IEEE Computer Society Press.

[58] GRUNDMANN, W. *et al.* Designing high performance CMOS microprocessors using full custom techniques. In *Proceedings of the 1997 Conference on Design Automation (DAC-97)* (June 1997), ACM/IEEE, pp. 722–727.

[59] HAUCK, S. Asynchronous design methodologies: An overview. Tech. Rep. TR 93-05-07, Department of Computer Science and Engineering, University of Washington, Seattle, 1993.

[60] HAUCK, S. Asynchronous design methodologies: An overview. *Proceedings of the IEEE 83*, 1 (Jan. 1995).

[61] HEMANI, A., MEINCKE, T., KUMAR, S., POSTULA, A., OLSSON, T., NILSSON, P., OBERG, J., ELLERVEE, P., AND LUNDQVIST, D. Lowering power consumption in clock by using globally asynchronous locally synchronous design style. In *Proc. ACM/IEEE Design Automation Conference* (June 1999), p. 873.

[62] HENNESSY, J., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.

[63] HOLLAAR, L. A. Direct implementation of asynchronous control units. *IEEE Transactions on Computers C-31*, 12 (Dec. 1982), 1133–1141.

[64] HUCK, J., MORRIS, D., ROSS, J., KNIES, A., MULDER, H., AND ZAHIR, R. Introducing the IA-64 architecture. *IEEE Micro* (Sept. 2000).

[65] HULGAARD, H. *Timing Analysis and Verification of Timed Asynchronous Circuits.* PhD thesis, Department of Computer Science, University of Washington, 1995.

[66] HWU, W., AND PATT, Y. N. HPSm, a high performance restricted data flow architecture having minimal functionality. In *Proceedings of the 13th Annual Symposium on Computer Architecture* (June 1986), pp. 297–307.

[67] Janik, K., Lu, S., and Mille, M. Non-stalling circular counterflow pipeline processor with reorder buffer. *U.S. Patent* (Dec. 2000). #6,163,839.

[68] Johnson, W. M. Super-scalar processor design. Thesis CSL-TR-89-383, Stanford University, Computer Systems Laboratory, June 1989.

[69] Josephs, M. B., and Yantchev, J. T. CMOS design of the tree arbiter element. *IEEE Transactions on VLSI Systems 4*, 4 (Dec. 1996), 472–476.

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

[71] Jouppi, N. P., and Wilton, S. J. An enhanced access and cycle time model for on-chip caches. Research report 93/5, DEC Western Research Laboratory, July 1994.

[72] Kahng, A. B., Muddu, S., and Vidhani, D. Noise and delay uncertainty studies for coupled rc interconnects. In *IEEE International ASIC/SOC Conference* (1999), pp. 3–8.

[73] Kearney, D., and Bermann, N. W. Performance evaluation of asynchronous logic pipelines with data dependant processing delays. In *Asynchronous Design Methodologies* (May 1995), IEEE Computer Society Press, pp. 4–13.

[74] Keller, R. M. Lookahead processors. *ACM Computing Surveys 7*, 4 (Dec. 1975), 177–195.

[75] Kessler, R. E. The Alpha 21264 microprocessor. *IEEE Micro 19*, 2 (March/April 1999), 24–36.

[76] Kohn, L., and Margulis, N. Introducing the Intel i860 64-bit microprocessor. *IEEE Micro 9*, 4 (1989), 15–30.

[77] Kolks, T., Vercauteren, S., and Lin, B. Control resynthesis for control-dominated asynchronous designs. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Mar. 1996).

[78] Kowka, K. J., and Galambos, T. Circuit design techniques for a gigahertz integer microprocessor. In *Proc. International Conf. Computer Design (ICCD)* (Oct. 1998), IEEE Computer Society, pp. 11–16.

[79] KUMAR, A. The HP PA-8000 RISC CPU. *IEEE Micro 17*, 2 (March/April 1997), 27–32.

[80] LAVAGNO, L., AND SANGIOVANNI-VINCENTELLI, A. *Algorithms for Synthesis and Testing of Asynchronous Circuits.* Kluwer Academic Publishers, 1993.

[81] LEUNG, A., PALEM, K. V., AND UNGUREANU, C. Run-time versus compile-time scheduling in superscalar (RISC) processors: Performance and trade-off. *Journal of Parallel and Distributed Computing 45* (1997), 13–28.

[82] LINES, A. M. Pipelined asynchronous circuits. Tech. Rep. Caltech-CS-TR-95-21, Computer Science Department, California Institute of Technology, 1998.

[83] LIPASTI, M. H., AND SHEN, J. P. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th International Symposium on Microarchitecture* (Paris, France, Dec. 1996), IEEE.

[84] LIPASTI, M. H., WILKERSON, C. B., AND SHEN, J. P. Value locality and load value prediction. In *Proceedings of the 17th conference on architectural support for programming languages and operating systems (ASPLOS)* (Oct. 1996), pp. 138–147.

[85] MANOHAR, R., LEE, T.-K., AND MARTIN, A. J. Projection: A synthesis technique for concurrent systems. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Apr. 1999), pp. 125–134.

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

[87] MARTIN, A. J. Asynchronous datapaths and the design of an asynchronous adder. *Formal Methods in System Design 1*, 1 (July 1992), 119–137.

[88] MARTIN, A. J., LINES, A., MANOHAR, R., NYSTROEM, M., PENZES, P., SOUTHWORTH, R., AND CUMMINGS, U. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI* (Sept. 1997), pp. 164–181.

[89] Miller, M., and Janik, K. Non-stalling counterflow architecture. In *4th Int'l Symp. on High-Performance Computer Architecture* (Feb. 1998), pp. 334–341.

[90] Molnar, C. E., Jones, I. W., Coates, B., and Lexau, J. A FIFO ring oscillator performance experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Apr. 1997), IEEE Computer Society Press, pp. 279–289.

[91] Monreal, T., González, A., Valero, M., González, J., and Viñals, V. Dynamic register renaming through virtual-physical registers. *The Journal of Instruction-Level Parallelism 2* (May 2000). http://www.jilp.org/vol2.

[92] Moore, S., Robinson, P., and Wilcox, S. Rotary pipeline processors. *IEE Proceedings, Computers and Digital Techniques 143*, 5 (Sept. 1996), 259–265.

[93] Moore, S., Taylor, G., Mullins, R., and Robinson, P. Channel communication between independent clock domains. In *Proceedings of the ACiD-WG Workshop* (Feb. 2001).

[94] Moore, S. W., Taylor, G. S., Cunningham, P. A., Mullins, R. D., and Robinson, P. Self-calibrating clocks for globally asynchronous locally synchronous systems. In *Proc. International Conf. Computer Design (ICCD)* (Sept. 2000).

[95] Morton, S. V., Appleton, S. S., and Liebelt, M. J. ECSTAC: A fast asynchronous microprocessor. In *Asynchronous Design Methodologies* (May 1995), IEEE Computer Society Press, pp. 180–189.

[96] Moshovos, A., Breach, S., Vajaykumar, T., and Sohi, G. Dynamic speculation and synchronization of data dependencies. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)* (June 1997), pp. 181–193.

[97] Moudgill, M., Pingali, K., and Vassiliadis, S. Register renaming and dynamic speculation: an alternative approach. Tr 93-1379, Department of Computer Science, Cornell University, Aug. 1993.

[98] MULE, A., SCHULTZ, S., GAYLORD, T. K., AND MEINDL, J. D. An optical clock distribution network for gigascale integration. In *IEEE International Interconnect Technology Conference* (June 2000), pp. 6–8.

[99] MULLER, D. E., AND BARTKY, W. S. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching* (Apr. 1959), Harvard University Press, pp. 204–243.

[100] MYERS, C. J., ROKICKI, T. G., AND MENG, T. H.-Y. Automatic synthesis and verification of gate-level timed circuits. Tech. Rep. CSL-TR-94-652, Stanford University, Jan. 1995.

[101] NANYA, T., TAKAMURA, A., KUWAKO, M., IMAI, M., FUJII, T., OZAWA, M., FUKASAKU, I., UENO, Y., OKAMOTO, F., FUJIMOTO, H., FUJITA, O., YAMASHINA, M., AND FUKUMA, M. TITAC-2: A 32-bit scalable-delay-insensitive microprocessor. In *Symposium Record of HOT Chips IX* (Aug. 1997), pp. 19–32.

[102] NANYA, T., UENO, Y., KAGOTANI, H., KUWAKO, M., AND TAKAMURA, A. TITAC: Design of a quasi-delay-insensitive microprocessor. *IEEE Design & Test of Computers 11*, 2 (1994), 50–63.

[103] NIELSEN, L. S., AND SPARSØ, J. An 85μW asynchronous filter-bank for a digital hearing aid. In *International Solid State Circuits Conference* (Feb. 1998).

[104] NOWICK, S. M. Design of a low-latency asynchronous adder using speculative completion. *IEE Proceedings, Computers and Digital Techniques 143*, 5 (Sept. 1996), 301–307.

[105] NOWICK, S. M., YUN, K. Y., AND BEEREL, P. A. Speculative completion for the design of high-performance asynchronous dynamic adders. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Apr. 1997), IEEE Computer Society Press, pp. 210–223.

[106] NYSTROEM, M. Pipelined asynchronous cache design. Tech. Rep. Caltech-CS-TR-97-21, Computer Science Department, California Institute of Technology, 1997.

[107] PALACHARLA, S., JOUPPI, N. P., AND SMITH, J. E. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (Denver, June 1997).

[108] PALACHARLA, S., P.JOUPPI, N., AND SMITH, J. E. Quantifying the complexity of superscalar processors. Technical report TR-96-1328, University of Wisconsin-Madison, Nov. 1996.

[109] PAVER, N. C., DAY, P., FARNSWORTH, C., JACKSON, D. L., LIEN, W. A., AND LIU, J. A low-power, low-noise configurable self-timed DSP. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (1998), pp. 32–42.

[110] PAVER, N. C., DAY, P., FURBER, S. B., GARSIDE, J. D., AND WOODS, J. V. Register locking in an asynchronous microprocessor. In *Proc. International Conf. Computer Design (ICCD)* (Oct. 1992), IEEE Computer Society Press, pp. 351–355.

[111] PEÑA, M. A., AND CORTADELLA, J. Combining process algebras and Petri nets for the specification and synthesis of asynchronous circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Mar. 1996), IEEE Computer Society Press.

[112] PLANA, L. A., AND UNGER, S. H. Pulse-mode macromodular systems. In *Proc. International Conf. Computer Design (ICCD)* (Oct. 1998), pp. 348–353.

[113] POPESCU, V., SCHULTZ, M., SPRACKLEN, J., GIBSON, G., LIGHTNER, B., AND ISAMAN, D. The metaflow architecture. *IEEE Micro* (June 1991).

[114] QUEEN, W. J. A 56 entry instruction reorder buffer. In *International solid state circuits conference* (San Francisco, CA, Feb. 1996), vol. 39, pp. 212–213.

[115] RABAEY, J. M., AND PEDRAM, M. *Low power design methodologies.* Kluwer Academic Publishers, 1996.

[116] RAU, B. R., AND FISHER, J. A. Instruction-Level Parallel processing: History, overview and perspective. *The Journal of Supercomputing 7*, 1/2 (May 1993), 9–50.

[117] REBELLO, V. *On the Distribution of Control in Asynchronous Processor Architectures.* PhD thesis, University of Edinburgh, 1996.

[118] RICHARDSON, W. F. *Architectural Considerations in a Self-Timed Processor Design.* PhD thesis, Department of Computer Science, University of Utah, UT, USA., Feb. 1996. CSTD-96-001.

[119] RICHARDSON, W. F., AND BRUNVAND, E. An architecture for a self-timed decoupled computer. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Mar. 1996), IEEE Computer Society Press.

[120] ROSENBERGER, F. U., MOLNAR, C. E., CHANEY, T. J., AND FANG, T. P. Q-modules - internally clocked delay-insensitive modules. *IEEE Transactions on Computers C-37*, 9 (Sept. 1988), 1005–1018.

[121] ROTENBERG, E., BENNETT, S., AND SMITH, J. E. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture* (Paris, France, Dec. 2–4, 1996), IEEE Computer Society TC-MICRO and ACM SIGMICRO, pp. 24–34.

[122] SAZEIDIS, Y., VASSILIADIS, S., AND SMITH, J. The performance potential of data dependence speculation and collapsing. In *Proceedings of the 29th Annual International Symposium on Microarchitecture* (Paris, France, Dec. 2–4, 1996), IEEE Computer Society TC-MICRO and ACM SIGMICRO, pp. 238–247.

[123] SEITZ, C. L. Ideas about arbiters. *Lambda 1*, 1, First Quarter (1980), 10–14.

[124] SEITZ, C. L. System timing. In *Introduction to VLSI Systems*, C. A. Mead and L. A. Conway, Eds. Addison-Wesley, 1980, ch. 7.

[125] SEIZOVIC, J. N. Pipeline synchronization. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Nov. 1994), pp. 87–96.

[126] SIMA, D. The design space of register renaming techniques. *IEEE Micro 20*, 5 (September/October 2000).

[127] SJOGREN, A. E., AND MYERS, C. J. Interfacing synchronous and asynchronous modules within a high-speed pipeline. In *Advanced Research in VLSI* (Sept. 1997), pp. 47–61.

[128] SMITH, J. E. Processor evolution: From red giants to white drawfs. In *Proceedings of the Workshop on Complexity-Effective Design (WCED'00, held in conjuction with ISCA-27)* (Vancouver, B.C., June 2000).

[129] SMITH, J. E., AND PLESZKUN, A. R. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture* (Boston, Massachusetts, June 17–19, 1985), IEEE Computer Society TCA and ACM SIGARCH, pp. 36–44.

[130] SMITH, J. E., AND PLESZKUN, A. R. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers 37*, 5 (May 1988), 562–573.

[131] SMITH, M. D., HOROWITZ, M., AND LAM, M. S. Efficient superscalar performance through boosting. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)* (New York, NY, 1992), vol. 27-9, ACM Press, pp. 248–259.

[132] SOHI, G. S. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers 39*, 3 (Mar. 1990), 349–359.

[133] SOHI, G. S., AND VAJAPEYAM, S. Instruction issue logic for high-performance interruptable pipelined processors. In *Proceedings of the 14th Annual Symposium on Computer Architecture* (June 1987), pp. 27–34.

[134] SOTELO-SALAZAR, S. *Instruction Scheduling in Micronet-based Asynchronous ILP Processors.* PhD thesis, Division of Informatics, University of Edinburgh, 2001. To be submitted.

[135] SPEC OPEN SYSTEMS GROUP (OSG). SPEC CPU95 Benchmarks. http://www.spec.org/osg/cpu95/.

[136] SPROULL, R. F., SUTHERLAND, I. E., AND MOLNAR, C. E. The counterflow pipeline processor architecture. *IEEE Design & Test of Computers 11*, 3 (Fall 1994), 48–59.

[137] STARK, J., BROWN, M. D., AND PATT, Y. N. On pipelining dynamic instruction scheduling logic. In *The Proceedings of the 33rd Annual International Symposium on Microarchitecture (MICRO'33)* (Dec. 2000).

[138] STEVENS, K., GINOSAR, R., AND ROTEM, S. Relative timing. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Apr. 1999), pp. 208–218.

[139] SUTHERLAND, I., AND FAIRBANKS, S. Gasp: A minimal fifo control. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Mar. 2001).

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

[141] TAYLOR, G. S., AND BLAIR, G. M. Reduced complexity two-phase micropipeline latch controller. *IEEE Journal of Solid-State Circuits 33*, 10 (Oct. 1998), 1590–1593.

[142] THEIS, T. N. The future of interconnection technology. *IBM Journal of Research and Development 44*, 3 (May 2000), 379–390.

[143] THISTLE, M. R., AND SMITH, B. J. A processor architecture for horizon. In *Proceedings of Supercomputing 88* (Orlando, Nov. 1988), pp. 35–41.

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

[145] TIWARI, V., SINGH, D., RAJGOPAL, S., MEHTA, G., PATEL, R., AND BAEZ, F. Reducing power in high-performance microprocessors. In *Proceedings of the 1998 Conference on Design Automation (DAC-98)* (Los Alamitos, CA, June 15–19 1998), ACM/IEEE, pp. 732–737.

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

[147] TRUONG, L. The VelociTI architecture of the TMS320C6xxx. In *Symposium Record of HOT Chips IX* (Stanford, California, Aug. 1997), pp. 55–63.

[148] TYSON, G. S., AND AUSTIN, T. M. Improving the accuracy and performance of memory communication through renaming. In *30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-30)* (1997), pp. 218–227.

[149] VAN BERKEL, C. H. K., JOSEPHS, M. B., AND NOWICK, S. M. Scanning the technology: Applications of asynchronous circuits. *Proceedings of the IEEE 87*, 2 (Feb. 1999), 223–233.

[150] VANBEKBERGEN, P., GOOSSENS, G., AND LIN, B. Modeling and synthesis of timed asynchronous circuits. In *Proc. European Design Automation Conference (EURO-DAC)* (Sept. 1994), IEEE Computer Society Press, pp. 460–465.

[151] VASSILIADIS, S., B.BLANER, AND R.J.EICKEMEYER. SCISM: A scalable compound instruction set machine. *IBM Journal of Research and Development 38*, 1 (1994).

[152] VERHOEFF, T. Delay-insensitive codes—an overview. *Distributed Computing 3*, 1 (1988), 1–8.

[153] WEISS, S., AND J.E.SMITH. Instruction issue logic in pipelined supercomputers. *IEEE Transactions on Computers C-33*, 11 (Nov. 1984), 1013–1022.

[154] WERNER, T., AND AKELLA, V. An asynchronous superscalar architecture for exploiting instruction-level parallelism. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Mar. 2001).

[155] WERNER, T., AND V.AKELLA. Counterflow pipeline-based dynamic instruction scheduling. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Mar. 1996), IEEE Computer Society Press.

[156] WILLIAMS, T. E., AND HOROWITZ, M. A. A zero-overhead self-timed 160ns 54b CMOS divider. *IEEE Journal of Solid-State Circuits 26*, 11 (Nov. 1991), 1651–1661.

[157] YAKOVLEV, A. Designing arbiters using petri nets. In *Proceedings of the 1995 Israel Workshop on Asynchronous VLSI* (1995), pp. 178–201.

[158] YAKOVLEV, A. Designing control logic for counterflow pipeline processor using Petri nets. Tech. Rep. TR 522, Dept. of Computing Science, Univ. of Newcastle upon Tyne, 1995.

[159] YAKOVLEV, A., PETROV, A., AND LAVAGNO, L. A low latency asynchronous arbitration circuit. *IEEE Transactions on VLSI Systems 2*, 3 (Sept. 1994), 372–377.

[160] YAKOVLEV, A. V., KOELMANS, A. M., AND LAVAGNO, L. High-level modeling and design of asynchronous interface logic. *IEEE Design & Test of Computers 12*, 1 (Spring 1995), 32–40.

[161] YANTCHEV, J. T., HUANG, C. G., JOSEPHS, M. B., AND NEDELCHEV, I. M. Low-latency asynchronous FIFO buffers. In *Asynchronous Design Methodologies* (May 1995), IEEE Computer Society Press, pp. 24–31.

[162] YEAGER, K. C. The MIPS R10000 superscalar microprocessor. *IEEE Micro 16*, 2 (1996), 28–40.

[163] YOAZ, A., EREZ, M., RONEN, R., AND JOURDAN, S. Speculation techniques for improving load related instruction scheduling. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA)* (May 1999).

[164] YUN, K., AND DOOPLY, A. E. Pausible clocking based heterogeneous systems. *IEEE Transactions on VLSI Systems 7*, 4 (Dec. 1999), 482–487.

[165] YUN, K. Y., BEEREL, P. A., AND ARCEO, J. High-performance two-phase micropipeline building blocks: double edge-triggered latches and burst-mode select and toggle circuits. *IEE Proceedings, Circuits, Devices and Systems 143*, 5 (Oct. 1996), 282–288.

[166] YUN, K. Y., BEEREL, P. A., VAKILOTOJAR, V., DOOPLY, A. E., AND ARCEO, J. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Apr. 1997), IEEE Computer Society Press, pp. 140–153.

[167] ZARKESH-HA, P., MULE, T., AND MEINDL, J. D. Characterization and modeling of clock skew with process variations. In *IEEE Custom Integrated Circuits Conference* (1999), pp. 441–444.