Performance-Directed Design of Asynchronous VLSI Systems

Samuel Scott Appleton
B.E.(Hons.)

A dissertation submitted in the
Department of Electrical and Electronic Engineering,
University of Adelaide, to meet the requirements for
the award of the degree of
Doctor of Philosophy.

THE UNIVERSITY OF ADELAIDE

27 August 1997

Abstract

Asynchronous system design has been subject to a resurgence of interest over the past decade due to the mounting problems with synchronous design approaches. Asynchronous systems eliminate or obviate many difficulties with large, complex VLSI systems encountered when using synchronous design techniques. Nevertheless, asynchronous design is still not widely accepted for a number of reasons, perhaps the most critical of which is the performance limitation imposed.

This thesis describes a new method for designing asynchronous systems that breaks with traditional techniques commonly believed to be essential. The method, called **Free-Flow Asynchronism** *(FeFA) eliminates all unnecessary signalling to open the performance bottleneck, while still retaining many inherent advantages of both synchronous and asynchronous design approaches. The method is based around an existing engineered design approach for two-phase asynchronous systems, Event Controlled Systems (ECS). ECS is shown to outperform existing asynchronous control methodologies by significant factors.*

Powerful techniques are developed to compose complex systems, describe solutions to problems that occur in implementations, and give performance figures based on simulation. A method for utilising variable latency in FeFA pipelines to significantly improve average performance is developed.

The method is demonstrated through two applications. An channel signalling system, which can be used in FeFA or traditional asynchronous systems, is described. This application demonstrates the basic techniques of the FeFA approach and design trade-offs. Amedo, an asynchronous microprocessor based around the DLX architecture, is then described. This serves to illustrate some of the more advanced aspects of FeFA system design, as well as demonstrating the potential performance benefits.

Contents

List of Figures

List of Tables

List of Abbreviations

DI	Delay Insensitive	Chapter 2, Section 1.2
SI	Speed Independent	Chapter 2, Section 1.2
BD	Bounded Delay	Chapter 2, Section 1.2
CSP	Communicating Sequential Processes	Chapter 2, Section 1.3.2
STG	Signal Transition Graph	Chapter 2, Section 1.3.1
2ϕ, TP	Two-Phase	Chapter 2, Section 1.1
4ϕ, FP	Four-Phase	Chapter 2, Section 1.1
DR	Dual-Rail	Chapter 2, Section 1.1.1
SR	Single-Rail	Chapter 2, Section 1.1.1
ST	Self-Timed	
PST	Pseudo Self-Timed	Chapter 2, Section 1.1.1
ECS	Event Controlled Systems	Chapter 3
TE	Temporal Equation	Chapter 3, Section 1.4
TS	Temporal Specification	Chapter 3, Section 1.4
EFG	Event Flow Graph	Chapter 3,Section 3
S-Pipe	State Pipeline	Chapter 4, Section 1.1
FIFO	First-In, First-Out (Buffer)	Chapter 4, Section 1.1
D-Pipe	D-Latch Register Pipeline	Chapter 4, Section 1.3
P-Pipe	Pulsed-Latch Register Pipeline	Chapter 4, Section 1.3
ECSTAC	Event Controlled Systems Temporally-Specified Asynchronous CPU	Chapter 5
ISA	Instruction Set Architecture	Chapter 5, Section 1.1
ICache	Instruction Cache	Chapter 5, Section 2.1
DCache	Data Cache	Chapter 5, Section 2.2
MU	Memory Unit	Chapter 5, Section 2.3

FeFA	Free Flow Asynchronous	Chapter 6
ISS	Issuing Stage	Chapter 6, Section 1
IE	Input-Event (Controller)	Chapter 6, Section 2
IOE	Input-Output-Event (Controller)	Chapter 6, Section 2
MDE	Matched Delay Element	Chapter 6, Section 4
STSC	Static Timing-Skew Compensation	Chapter 6, Section 4.2
MR	Multi-Rate (Pipeline)	Chapter 6, Section 6
NAC	Non-Acknowledging Communication	Chapter 7, Section 2
RTL	Register Transfer Language	Chapter 8,Section 1.3
CPI	Cycles Per Instruction	Chapter 8,Section 1.4
LRB	Load Register Buffer	Chapter 8, Section 2
$\mathcal{T}_g$	Gate Delay	Appendix C

Declaration

This thesis has been submitted to the Faculty of Engineering at the University of Adelaide for examination in respect of the Degree of Doctor of Philosophy.

This thesis contains no material which has been accepted for the award of any other degree or diploma in any University, and to the best of the author's knowledge and belief contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

The author hereby consents to this thesis being made available for photocopying and for loans as the University deems fitting, should the thesis be accepted for the award of the Degree.

Sam Appleton
27 August 1997.

Acknowledgments

Thanks to one of my greatest friends, Shannon Morton. Shannon convinced me to return to University to do my Ph.D. degree and subsequently became a lifelong friend, as well as doing some truly brilliant work. A well-spring of inspiration in every situation imaginable, Shannon taught me to look around preconceived notions and prejudices to extract the true essence of an idea.

My supervisor, Michael Liebelt, was one long-time believer, even when things turned dark. Mike kept me in a positive mindset throughout my postgraduate days, even when some great shining hopes collapsed in a screaming heap, and encouraged me when I discovered something that turned out to be quite spectacular. Mike was also the main force in obtaining the much needed funds for both ECSTAC and another up-and-coming microprocessor. Thanks, Mike.

To all the friends I made over the years, thanks for being there! Dr. Pucknell for providing me with good advice and supervision in my final undergraduate year, Andrew Johnson for preliminary work, and all the "Digital Lab" crew – Nick Betts, Tim Shaw, Braden Phillips, and Nasser Asgari — with special guests celebrities Ali Moini, Richard Beare, Said Al-Sarawi, Nozar Tabrizi, Kiet To, the big man Stéphane Lefrere, and late addition Chris Howland. Special thanks are due to Andrew Beaumont-Smith for endless discussions on circuits, architectures and processes, and Andrew Blanksby for extraordinary surfing expeditions and unique humor. My thanks also to all of the final-year project students who worked on various aspects of the ECS project over the past few years, as well as the technical staff in the department – Geoff Pook, Mark Bailye and Gordon Allison, and computing legends David Bowler and Nick Kerr. Special mention should also be made of my Korean connection at Seoul National University – Wonchan Kim and all the great students in ISDL, especially Gyudong Kim, Jungwook Yang, Daejong Kim and Ook Kim.

Thanks to my Mum, Dad, and Ned for being family and more. Your support was, and still is, a great help. And finally, to Bec, thanks for your love, support, and unbounded enthusiasm — it made so much possible.

Sam.

Chapter 1

Introduction

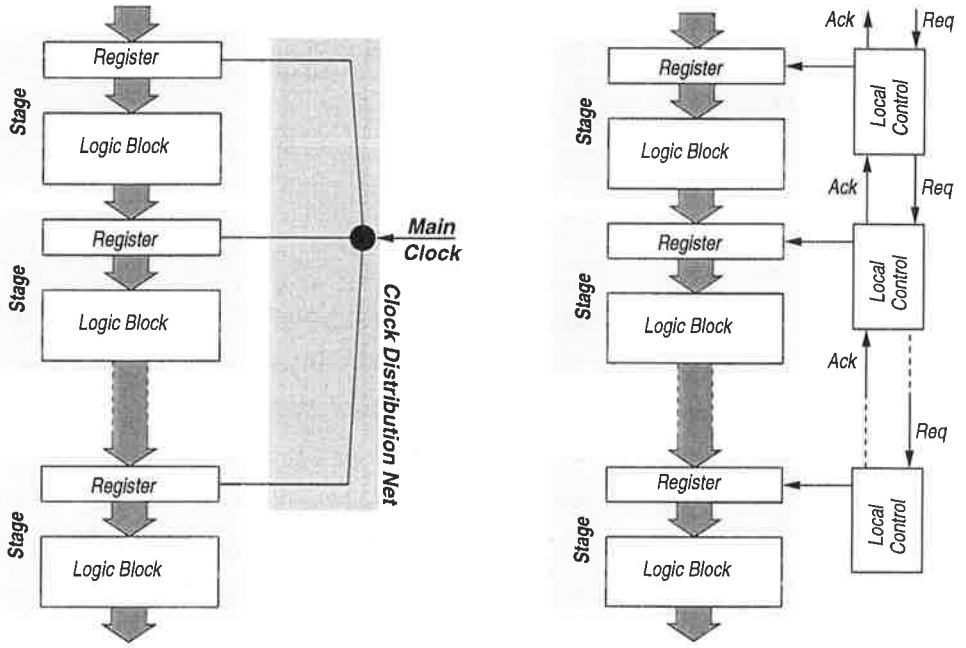
DIGITAL systems designed using asynchronous techniques have long been considered the eventual replacement for the currently dominant synchronous design paradigm. However, the lack of acceptance by the VLSI design community at large indicates that much of the work done to date does not mandate the switch to the asynchronous paradigm. The primary issue involved is that of *performance* — asynchronous system performance is typically much lower than that for a synchronous system implemented in the same technology. Since performance goals are usually integrated with functional goals, the performance problems in building asynchronous systems hinder their wider acceptance to a significant degree, even though they may offer real advantages in power consumption and some design issues.

This work describes a method obviating the severe performance disadvantage incurred by switching to an asynchronous design style, while retaining the truly important advantages of asynchronous systems that make them so appealing. The *Event Controlled Systems* methodology for asynchronous design, developed by Morton [Mor97], has shown the performance potential of asynchronous systems when engineered design practices are vigorously pursued, and a similar direction is adopted in this work. This thesis takes the desire for improved performance to its logical conclusion, and may well be the limit of asynchronous pipeline performance in a given technology.

1 Asynchronous Systems

Communication methods for synchronous and asynchronous systems are shown in Figure 1.1. Synchronous systems operate by synchronising data transfer between or on the edges of a

global clock signal, shown in Figure 1.1(a). The incoming CLOCK signal is distributed through a clock network to registers, which separate logic blocks into stages. Since the global CLOCK signal sequences and controls all communication actions, there is little or no inter-stage control necessary. Synchronous pipeline throughput is limited by the worst-case delay of any stage, even if this case occurs relatively rarely.



(a) Synchronous Communication

(b) Asynchronous Communication

Figure 1.1 Synchronisation Mechanisms. Synchronous schemes, (a), for pipeline synchronisation require a clock used as a global timing reference. Asynchronous systems, (b), use localised signalling, thereby eliminating the global clock.

Asynchronous systems operate by performing local synchronisation between communicating stages. This typically involves at least two signals, REQUEST to send a communication request, and ACKNOWLEDGE, to acknowledge that the request and data have been successfully received, shown in Figure 1.1(b). In such systems, *local controllers* replace the global clock signal in controlling the inter-stage registers. This method of synchronisation is widely cited as the eventual successor of the currently dominant *synchronous* design style for a variety of reasons [Mar92]. However, asynchronous design has a number of inherent problems and a number of current design problems which continue to hinder wider acceptance of the approach. Some of the major points of interest concerning both synchronous and asynchronous approaches shall now be examined.

1.1 Modularity

Asynchronous modules communicate with other modules with all timing and synchronisation information transfered at the module interfaces. Therefore, as long as two modules meet the interface requirements they will operate together correctly, regardless of the relative speeds of the two modules. This also implies that an asynchronous system can be incrementally improved by replacing slow modules with faster ones — as long as the new module is functionally equivalent and has the same interface, the system will operate correctly.

1.2 Power Consumption

The distributed, locally-controlled nature of asynchronous control causes the power consumption of the system as a whole to be more evenly distributed in the time domain, as opposed to synchronous designs which draw most of their power at the clock switching intervals. In addition, asynchronous systems inherently *power-down* inactive units, causing the power consumption of a chip with varying computational load to drop considerably. The elimination of the global clock also means that there is no longer any need to generate a very fast, tightly-controlled signal and distribute it to all parts of the chip, reducing power consumption further — clock distribution itself can account for up to 40% of total chip power [Bou96].

1.3 Lack of Global Synchronisation Difficulties

The most widely touted advantage of asynchronous design is the lack of global synchronisation issues in the design of large, complex systems. As several recent high-end CMOS designs have illustrated [BBB+95, LCT+95, Yea96], the design of the clocking network is becoming increasingly sophisticated and much more difficult as process technology progresses. Interconnect wiring of the complexity used in clock distribution now demands full RLC extraction [VYSS96], and special techniques [Ell96, EPR97, Fri95] are required to ensure the clock operates correctly with minimum skew. Asynchronous approaches solve this problem by definition, since there is a complete absence of global synchronisation control. Instead, all timing constraints are now localised in asynchronous modules and sub-systems, theoretically making timing verification easier since all verification of timing issues can be done at the sub-system and module level.

1.4 Operating Condition Resilience

The synchronisation problem has encouraged many research efforts into systems which require only a minimum of timing verification (a small timing verification requirement exists in order that a non-trivial circuit can be designed), thus creating devices which operate over almost any temperature, voltage, and process variation condition, no matter how the design was implemented.

This means that any change in module, gate or system parameters has no effect on the functionality of the system as a whole — it will always perform correctly. In addition, a "change for the better" in parameters produces a corresponding increase in performance – a system-wide improvement in operating conditions results in a system-wide improvement in performance, since the system performance is determined by the system, and not by an externally-generated synchronisation signal.

Asynchronous systems which do not operate independently of all system parameters can still be partially resilient to parameter variation, Modules whose timing is modelled by delays will operate correctly over wide variations in operating temperature and voltage, depending on the design style, and also exhibit the corresponding performance variations.

1.5 Synthesis and Verification

As asynchronous modules communicate over a well-defined local interface that can be made independent of timing, the design of asynchronous systems is widely facilitated by using formal program design techniques to describe inter- and intra-module communication and function. This approach results in systems that are correct-by-construction and guaranteed to work under any conditions. In addition, because most asynchronous approaches operate without timing constraints, their functionalitity can be checked by formal techniques and *verified* without need for a large amount of simulation.

1.6 Performance

The change from global to local synchronisation can have a severe impact on the performance attainable by an asynchronous system. The delay overhead of the local inter-stage communication adds to the critical path logic delays in the circuit, impacting on the cycle time achievable by an asynchronous implementation. This is perhaps the greatest impediment to the acceptance of asynchronous mechanisms, even considering all the advantages that the approach embodies. Traditional asynchronous approaches may have roughly comparable area and significantly lower power consumption, but the performance penalty can be incredible – anywhere from 20% to 200% slower would not be unusual.

1.7 Complexity

Although the number of gates needed to implement the local communication action in each asynchronous module may be quite small and negligible with respect to the area and power consumption of the datapath, there is an additional penalty for switching to the asynchronous paradigm that is inherent to the approach. Many tasks which are trivial or easy to perform in synchronous design, such as data forwarding and backwarding, are much more complex in asynchronous approaches due to the distributed, non-synchronised nature of the computations.

In addition, to eliminate parameter variations many methodologies advocate the use of *dual-rail* computation, in which both the result function and the complementary result function are computed. The area overhead for such an approach is up to two times the area of an equivalent *single-rail* implementation. This complexity would, in practice, severely limit the use of asynchronous systems for no other factor than cost.

1.8 Design Tools

VLSI design flows rely heavily on the use of sophisticated, powerful tools for automating and checking design tasks. These tools are predominantly focussed on providing support for synchronous design. There are some powerful tools for the design of limited-size asynchronous circuits, but in general the state of design automation for asynchronous design remains bleak, primarily because of a lack of commercial interest in the approach.

1.9 Test

The distributed nature of asynchronous control is problematic when test issues are considered. A synchronous system may be stopped or clocked at a lower rate than normal to facilitate test of the circuits on the device. However, asynchronous systems cannot easily be stopped or controlled in this manner, and thus global control generally required for test purposes becomes difficult. In addition, the larger number of storage elements used by asynchronous approaches increases the size of the test coverage problem.

1.10 Real-Time Constraints

The ability of synchronous controller to meet real-time constraints is a direct result of the *timing knowledge* that the clock embodies – by counting the number of operations and the number of clock cycles each takes, the computation time of a given algorithm can be calculated. The lack of this timing information in the asynchronous paradigm means that real-time constraints are hard to quantify and harder to guarantee, and little work has been done to address this problem. In addition, performance determined by the silicon implies system to system performance variation (i.e. not consistent from one device to the next, nor consistent with other devices in slightly different operating environments), which exacerbates the real-time constraint problem considerably.

2 Thesis Outline

This thesis develops an asynchronous approach fundamentally different from existing asynchronous approaches, and focuses on achieving near-synchronous performance while retaining

the traditional advantages of asynchronous techniques — power spectrum improvement, automatic power-down of inactive systems, no global routing or timing issues, and a good degree of operating condition resilience. This approach is described in Chapter 6. Figure 1.2 shows the flow of concepts in the thesis.

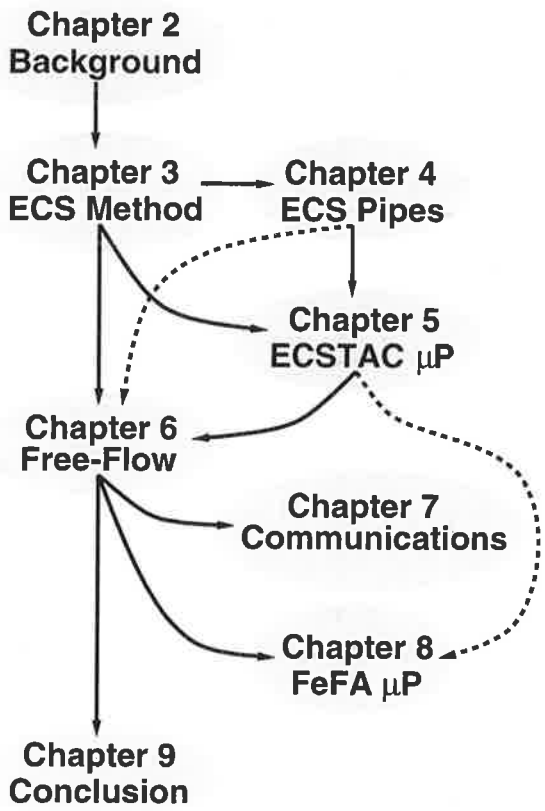


Figure 1.2 Thesis Flow. Normal Lines show strong correspondence between concepts presented in chapters, while dashed lines show weak correspondence and indicate *learning experiences* for later work.

Chapter 2 provides a detailed analysis of the present state-of-the-art in asynchronous system design, ranging from fully speed-independent systems to bounded-delay systems. The concepts, models and techniques underlying the various methods of asynchronous design are examined, and hardware systems implemented using the various approaches are described. Some related work is described, the asynchronous microprocessor STRiP [Dea92] and the STARI [Gre93] communication technique.

Chapter 3 presents a design methodology for two-phase asynchronous systems using a bounded-delay timing model. This approach, termed *Event Controlled Systems* (ECS), is an engineered

approach to design, rather than a correct-by-construction approach. The design methodology is described, along with the gates and models required to compose control networks. Modifications to both the notation and the operational mechanisms are undertaken to improve consistency. The ECS approach forms the basis of many ideas which lead to the development of the new asynchronous method described in this thesis. Chapter 4 explores the design and performance of ECS pipelines. The performance advantage of ECS pipelines against other approaches is demonstrated through circuit simulation, showing greater than 50% improvements in cycle time over current asynchronous approaches. The ECS notation is extended with the concept of *interface timing*, and methods for the test of ECS pipelines are described.

Chapter 5 describes the major issues in the development of a prototype asynchronous microprocessor, ECSTAC, designed using the two-phase ECS approach. The architecture of the processor and the decomposition process employed are described, and relevant logical and circuit issues are discussed. This design raises and analyses a number of critical issues which form part of the reasoning behind a new methodology. The results of the chip testing are also summarised.

Chapter 6 develops a new methodology for asynchronous system design termed *Free-Flow Asynchronism* (FeFA). The motivation and basis for the development of *Free-Flow Asynchronous* systems are described. FeFA uses techniques and models closely related to the *Event Controlled Systems* approach. A complete exposition of the FeFA method is developed, including pipeline composition methods and operating constraints. The critical issues affecting the implementation of these system at the VLSI level are also examined, and the performance of FeFA implementations against the *Event Controlled Systems* approach is compared.

Chapter 7 describes one application of the FeFA approach — channel communication between asynchronous systems. This application, called *Non-Acknowledging Communication* (NAC), was the forerunner of the Free-Flow approach. Chapter 8 describes the main application developed using FeFA, a 32-bit RISC microprocessor based on the DLX architecture [PH96]. A basic architecture is developed, then extended with modern microarchitectural improvements and some unique architectures from FeFA. The performance advantages of this architecture are demonstrated through detailed simulation of benchmark programs.

Chapter 9 concludes the work and provides suggestions for future development of the ECS and FeFA approaches.

3 Contribution and Concluding Remarks

The primary contribution of this thesis is in the development of a new asynchronous design style, called *Free-Flow Asynchronism* (FeFA). This methodology allows high-speed pipelined asynchronous systems to be designed and implemented, and retains most of the advantages of both synchronous and asynchronous design domains. The design approach is highly engineered and much work would have to be done in order to automate design procedures, although design work could be eased by using the block architectures described in this thesis. The design of the free-flow microprocessor, Amedo (Chapter 8), is wholly the work of the author, and contains several architectural innovations uniquely suited to the exploitation of timing in a *free-flow* architecture.

The author also made significant contributions to the development of the *Event Controlled Systems* methodology, although this work is largely attributable to Morton [Mor97]. The development of the cache architectures and the basic asynchronous pipeline control mechanism used in the prototype microprocessor, ECSTAC, are wholly attributable to the author. In addition, the author has expanded the ECS method with work on notation, and developed a number of pipeline architectures using *Event Controlled Systems* as well as basic testability strategies for these systems, described in Chapter 4, and obtained preliminary test data on the ECSTAC microprocessor.

Chapter 2

Background

THIS chapter discusses the existing body of work upon which this thesis is based. The concepts and methodologies underlying asynchronous design practices are presented and analysed, and hardware systems implemented using these approaches are described. Prior work related to this thesis is then described.

1 Asynchronous System Design

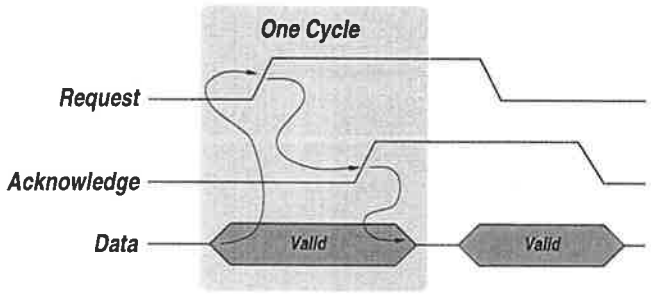
The field of asynchronous design is diverse, and excellent reviews on the topic abound [Hau95, BS95b, GJ90]. The taxonomy of asynchronous systems, and how they relate to the myriad implementation decisions that must invariably be made during the design process, will be the primary concern of this section.

1.1 Concepts

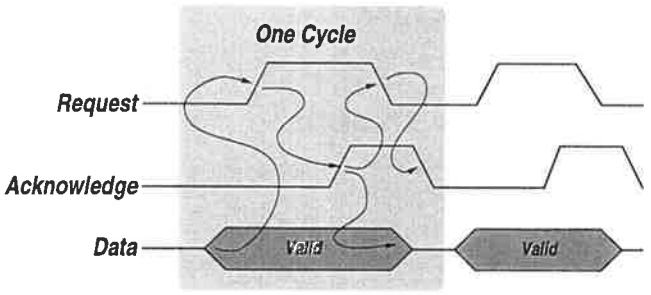
An asynchronous system replaces a global synchronising clock signal with a number of local communicating modules that implement system function. The method that these modules use to communicate is set by some transmission protocol, and usually uses a *request* and *acknowledge* system.

One transmission protocol uses two-phase or *transition* signalling. Two-phase signalling systems initiate a communication by transitioning the *request* line, and the receiver responds (when ready) by transitioning the *acknowledge* line. The two-phase protocol is illustrated in Figure 2.1(a). Note that the signalling is independent of the current logical state of the communication lines. A classic example of the use of two-phase signalling is Sutherland's *micropipelines* [Sut89].

Four-phase signalling systems initiate a communication by taking the *request* line high, and the receiver responds by taking *acknowledge* high (in general). The sender and receiver then take *request* and *acknowledge* low via some defined protocol. The four-phase protocol is illustrated in Figure 2.1(b). The return-to-zero phase of this protocol can be used for transferring more control information between the communicating modules [FL96].



(a) Two-Phase Communication Protocol



(b) Four-Phase Communication Protocol

Figure 2.1 Asynchronous Communication Protocols. The arrows indicate the timing flow and dependencies.

One other recently proposed system is the single-wire protocol [BB96], in which the sender has control over one phase of the communication, and the receiver has control over the other phase, all on a single wire. For example, the sender may take the communication wire high, and the receiver acknowledges by taking the communication wire low. The full benefits of this protocol remain to be fully explored, since the protocol does not give a definitive speed increase and only a modest improvement in area compared to the original Tangram program translation [Ber92b].

1.1.1 Control/Data Encoding

Asynchronous systems, due to their request/acknowledge protocols, will require some form of *completion* signal from the datapath components, which indicates that the current computation has finished. This information can either be explicitly embedded in the datapath, or can be approximated separately to the datapath.

Dual-Rail (DR) encoding

The most common method in which datapaths can signal their own completion is by using *dual-rail* computations. In this system, logic can be in one of three possible states : WAITING or RESET, VALID-ONE, or VALID-ZERO. The valid-one or valid-zero signals explicitly encode completion data because they are separate from the *waiting* state. These three states require two wires to encode (with the fourth possible state typically assumed *not allowed*), with a typical encoding [WH91] shown in Table 2.1.

DR encoding		
State	Wire-0	Wire-1
Reset	0	0
Valid-Zero	1	0
Valid-One	0	1
Invalid/Error	1	1

Table 2.1 Dual-Rail Encoding Scheme.

This scheme is very effective on a local level because arriving signals can self-synchronise to start computations, and computations require no external interaction or timing control. Thus dual-rail logic is highly effective at implementing logical functions while embedding timing information directly into dataflow. The one disadvantage of dual-rail computations is the area overhead — because both the VALID-ZERO and VALID-ONE signals must be generated, roughly two times the silicon area is required to implement a given function because both the result and the complement of the result must be computed.

When logic is separated by latches in a pipeline structure, the latch control must ensure that *all* the input signals are valid before *closing* the latch. This scenario is illustrated in Figure 2.2. Note that the very wide AND gate is usually very slow, which limits the speed at which this circuit can operate. Morton [Mor97] explored some novel schemes for very wide fan-in completion trees using semi-precharged techniques, which improved detection time at the expense of dynamic power consumption.

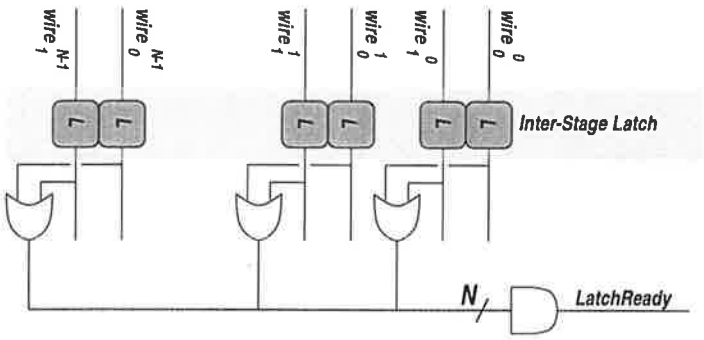


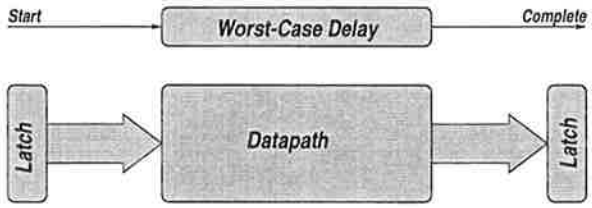
Figure 2.2 Dual-Rail Inter-Stage Latching. General required scenario for completion from an inter-stage latch using DR encoding – all bits must be checked for completion and ANDed to produce *LatchReady*.

One alternative to this expensive approach (in terms of area and time) is available when the depth of the datapath is typified by one particular bit (either because of a flat computation like carry-save addition [WH91], or a static worst-case path) — this then requires only one OR operation on the critical DR bit-pair, coupled with a timing assumption that this bit-pair always completes last, which would be verified either by simulation or analysis of the topology of the circuit. If neither of these is possible, a full OR-tree will be required. Another alternative is to use a delay model for the datapath, however, these models are normally associated with the use of single-rail encoding.

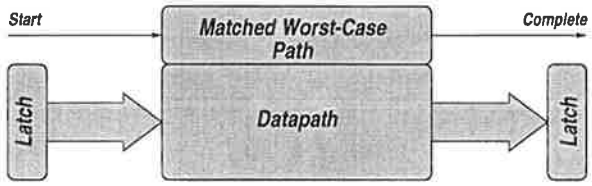
Single-Rail (SR) encoding

Single-rail encoding uses only one signal to represent one bit of information, and thus cannot encode completion or timing information on its own (if a signal is assumed to encode two states only). This is the normal model for synchronous design. SR models are also known as *bundled-data* techniques, since the datapath signals are *bundled* together, conceptually, to form the output together with some *bundle* completion signal (which goes valid at the same time or later than the data computation completes, known as the *bundled data constraint*). There are three alternatives for generating this completion signal — worst-case delay, matched-path, and pseudo-self-timing, shown in Figure 2.3.

The Matched Path model, shown in Figure 2.3(b), only differs from the worst-case delay model of Figure 2.3(a) in that the Matched Path is an exact circuital duplicate of the worst case path, whereas the delay model is an approximation based on simulation. Matched Path models are useful where the worst-case path is readily identified and easy to replicate, for example in equality comparators [AML94a, App96], and are potentially faster than delay models because they do not need margining (delay models must be slightly over-designed to account for uncertainty in simulation models and parameter variation effects, an issue to be explored in more depth in Chapter 6).



(a) Worst-Case Delay Model



(b) Matched Path Model

(c) PST model

Figure 2.3 Single-Rail Completion Signalling Methods.

Pseudo self-timing (PST) is an alternative self-timing approach, similar to dual-rail computations, that allows full datapath completion to be generated at a much lower cost than traditional DR methods [Mor97]. PST duplicates the path taken by critical signals in a *valid* network, which is observed for completion status. This results in circuits similar to matched-path-based techniques, but able to take advantage of data dependencies in completion timing.

Exotic Methods

SR circuits are more area-efficient than DR circuits, but SR circuits do not indicate completion based on datapath activity, instead using approximation techniques or mirroring to generate the desired signal.

Current-sensing techniques [DDH94] have been proposed to provide true monitoring of the SR datapath and indicate completion upon cessation of datapath activity. The basic technique for current-sensing completion detection (CSCD) is shown in Figure 2.4.

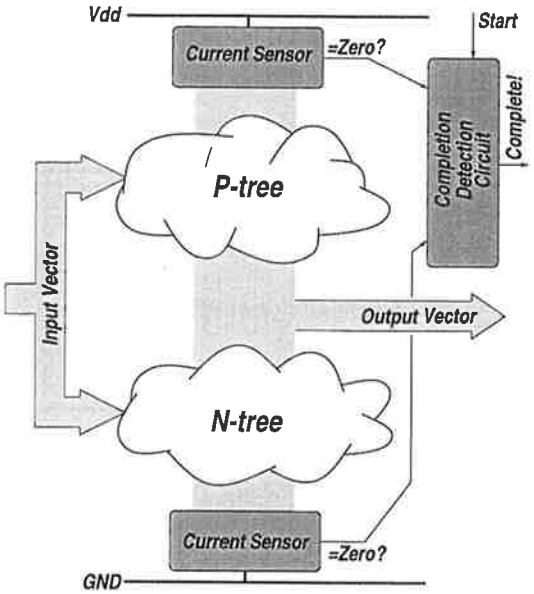


Figure 2.4 Current-Sensing Completion Detection.

Current-sensors detect when the P and N-tree parts of the logic block have stopped drawing current, and provided the block is active (indicated by the *Start* signal) indicate completion. The cessation of activity within both blocks indicates completion because no further signal transitions are occurring. However, the design of the current-monitoring block is, in general, a difficult analog circuit design problem. Other methods have been proposed that simplify design issues [GJ95, GMK96], but the technique remains within the realm of exotic circuit implementations.

1.2 Models

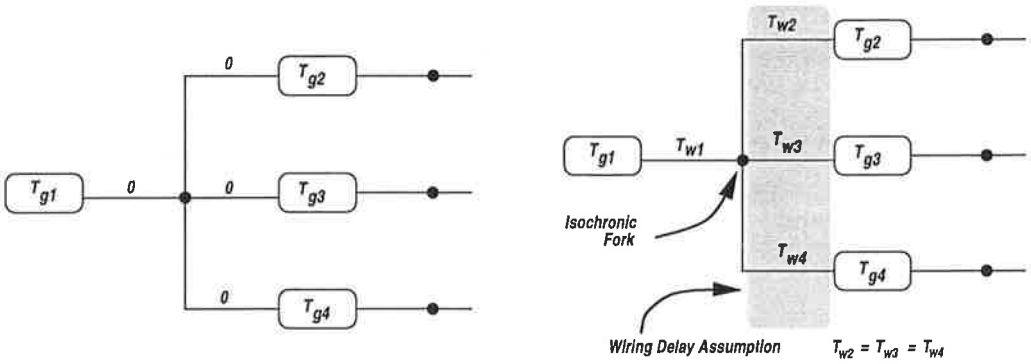
Asynchronous systems are often classified according to the assumptions they make about gate and wire delays in implementations. There are three classes of assumptions that can be made — Delay Insensitive (DI), Quasi-Delay Insensitive (QDI) or Speed-Independence (SI), and Bounded-Delay (BD).

1.2.1 Delay Insensitive Circuits

Delay Insensitive circuits, as their name suggests, operate correctly regardless of any delays in either wires or gates. Martin showed that, in general, only a very limited class of system implements total delay insensitivity [Mar90b], and unfortunately these circuits are not very useful. Thus, delay-insensitivity requirements are typically weakened to *quasi*-Delay Insensitivity [Mar90a], more well known as *Speed-Independence.*

1.2.2 Speed Independent Circuits

The Speed-Independent (SI) delay model, assumes that the delays in gates are potentially unbounded, and the delays in wires are zero, as shown in Figure 2.5(a).



(a) Speed Independent Delay Model

(b) Quasi-Delay-Insensitive Delay Model

Figure 2.5 Speed Independent Delay Models. The SI model, (a), assumes zero wire delays, while the QDI model, (b), assumes equal wire delays after an *isochronic fork*. Both assume potentially unbounded gate delays.

The QDI model is functionally equivalent to the SI model because it assumes equal wiring delays after an *isochronic fork*, the position on the gate output wire where the signal splits to feed two or more gates. As gate delays are assumed unbounded, these wiring delays can be pushed back into the (already potentially unbounded) gate delays, resulting in a model functionally identical to Speed Independence. The QDI model with *isochronic fork* is shown in Figure 2.5(b), for which some physical design constraints have been shown to be necessary [Ber92a].

1.2.3 Bounded Delay Circuits

Bounded-delay (BD) circuits assume that the delays in all gates and wires of a system are known (to some approximation), or at least *bounded*. Some early work [Huf64] explored asynchronous bounded-delay circuits operating in *fundamental mode*, where the inputs are not allowed to change until the all nodes have stabilised from the effect of the last input change. Later work expanded fundamental mode operation [Ung69, Ung97] to include multiple-input changes and I/O mode (where any input change must only wait to observe one output change, as opposed to the total internal stabilisation model of fundamental mode, before applying a new input change).

An example of a bounded-delay circuit is shown in Figure 2.6. This circuit would be used as a pulse generator on the rising edge of *EdgeSignal* by making a timing assumption that the delay from *EdgeSignal* to A was less than that from *EdgeSignal* to B. However, if the delay from *EdgeSignal* to A is greater than that from *EdgeSignal* to B, then we may get a pulse on the falling edge of *EdgeSignal*. Thus, the behaviour of the circuit is dependent on component delays, and would fail speed-independent criteria, but may be entirely satisfactory in a bounded-delay methodology.

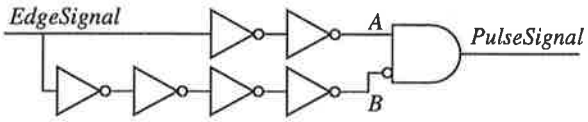


Figure 2.6 Bounded-Delay Circuit Example. The circuit is a pulse generator.

1.3 Methodologies

There are several different methodologies for asynchronous design. These approaches can be grouped into one of three generic classes. Graph-based approaches use a petri-net or graphical description of circuit or system function, and use varying levels of transformation techniques to obtain a circuit that satisfies the specification. Language-based methods produce a similar result, but start with a specification of the circuit using a textual description. Engineered approaches rely upon the skill of the designer to effect the transformation from specification (which may not be well-formalised) to implementation. This process may be guided or totally ad-hoc.

1.3.1 Graph-based Approaches

Most graph-based approaches derive from Petri Nets [Mur89, Pet81]. Petri Nets are general modelling tools that are suitable for a wide range of system description activities, and have been widely adopted for use in describing asynchronous circuits [Mis73, nC96].

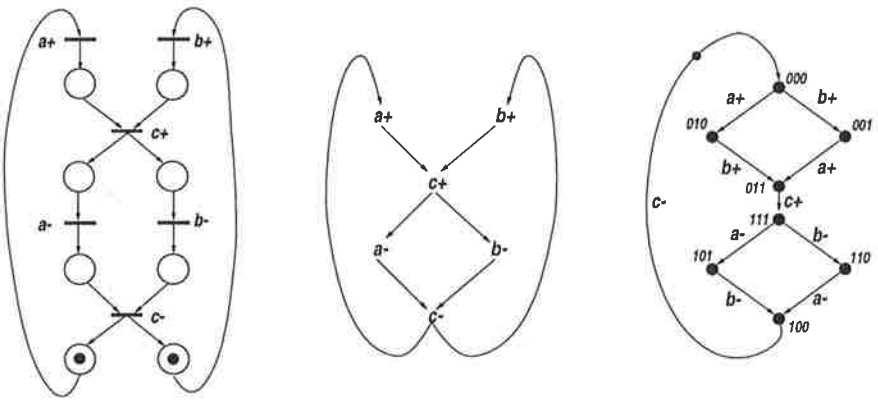
Even though graph-based approaches are extraordinarily good at representing the behaviour of small to medium size circuits, they can become unwieldy when dealing with large circuits (such as datapaths). Current work [DW95, FD96, YBA96] tends to use graph techniques to describe small control circuits, with other techniques being used for the datapath and system design tasks.

I-nets

I-Nets, or *Interface Nets*, allocate signals of a system to *transitions* of a petri-net, with the places determining signal flows [BS95b, Chapter 15]. Once specified, the states on the I-Net can be encoded and an ISG (*Interface State Graph*) formed, and a logic implementation realised using standard mapping methods. I-Nets are restricted to single-input-change systems, and become quickly unworkable for large circuits.

Signal Transition Graphs(STGs)

Signal Transition Graphs, developed by Chu [Chu87], are directed Petri Nets. The STG itself only specifies the behaviour of the system, and the STG itself is transformed into a structure more amenable to implementation. STGs have proved to be very useful and form the basis of a wide range of work [KKTV94, BS95b, Hau95]. An example of a STG is shown in Figure 2.7



(a) Petri Net Model (b) STG form (c) State-Variable Enumerated STG

Figure 2.7 STG examples. This STG, implementing a Muller-C gate, can be enumerated to lead to a circuital description.

STGs make the expression of concurrency relatively simple, as independent branches of the STG are separate from any others. STGs are restricted to *safe* and *live* nets [Chu87] (such concepts can be applied to ordinary Petri Nets [Mur89]). Chu discusses methods for optimisation of the net structure that are applied before the state coding step (see Figure 2.7(c)), which result in much simpler implementations. Other techniques also exist for optimisation of the final state encoding [VCGM90, VLGdM92, VGCM92]

Other Methods

Change Diagrams (CDs) [KKTV92] employ a description similar to STGs, but change the description of firing rules by employing different classes of arcs (which connect places to transitions and vice versa). CDs remove some of the restrictions created by the STG environment.

An extensive tool set for designs using STGs has been integrated into SIS (*Sequential Synthesis System*) [SSL+92], which automates a large amount of logic minimisation and implementation effort. Other tools for verification of speed-independence such as FORCAGE [KKTV94] and VERDECT [EB95] have been described based on STG-like approaches.

STG-like graphs and descriptions are used extensively. Myers [Mye95] described an approach using STG-like constructs called *Orbital Nets* that are used for the description and synthesis of timed asynchronous circuits. Hulgaard [Hul95] develops algorithms for the timing analysis of restricted class of Petri Nets with added timing data.

1.3.2 Language-Based Approaches

The majority of work on language-based methods is based on Hoare's *Communicating Sequential Processes* (CSP) [Hoa78, Hoa85], which in turn builds upon Dijsktra's *guarded commands* [Dij75]. CSP was designed to model parallelism in formally-described computer programs, and included specific constructs to formalise communications between program modules. The similarity to specification of communicating, parallel hardware systems did not go unnoticed. CSP was also used as a basis for the implementation of a new programming language, Occam [Inm88], which explicitly described parallelism in programs.

Language-based methods have proved to be very durable because they scale well with the size of the system being implemented. Even though language methods may not be as *intuitive* as graph methods, the compactness and power of the language constructs make these methods more suitable for larger designs.

Communicating Hardware Processes

Martin's work on Communicating Hardware Processes [Mar90a], or CHP, is perhaps the best known example of a language based approach based on CSP. The specification is first implemented as a concurrent CSP-like program which communicates over one-way channels. An example of such a program is shown in Figure 2.8.

This process *BUF* accepts a communication on channel A (indicated by the ? symbol), placing it in variable x, and sending the variable x onto the channel B (indicated by the ! symbol). The $*$ indicates an endless process loop. Note that the variables have no *capacity* — the x!B process must complete before a new x?A process can initiate. The initial CSP

*BUF = *[x?A,x!B]*

Figure 2.8 CHP One-Place Buffer.

program is then transformed into a set of *handshaking expansions*, which more closely mirror their final implementation in VLSI. This form is then transformed to *production rules*, which are simple transition specifications on boolean variables which can be readily implemented in CMOS VLSI, for example

$$\begin{aligned} \neg a \wedge \neg b &\rightarrow o\downarrow \\ a \wedge b &\rightarrow o\uparrow \end{aligned}$$

for a C-element (with a and b as inputs, and o as the output). These production rules can be tested for *stability* (checking that an enabled signal stays true until all rules it affects have fired) and *non-interference* (checking that the $\uparrow$ enabling guard and the $\downarrow$ enabling guard are mutually exclusive) [Coo93]. Once transformed and optimised, these rules can be implemented directly in CMOS VLSI switch-level circuits. All levels of the program transformation preserve the *semantics* of the original CHP program, and the method produces speed-independent implementations.

Martin's work has been significantly extended with developments in automation [Bur88, Coo93], performance analysis [Bur91] and optimisation [Lee95], energy analysis [Tie95] and testing [Haz92]. The robustness of the approach has been well-demonstrated through fabrication (see Section 1.4).

Trace Theory

Trace Theory [Sne85] is another method for describing concurrency in systems that has been applied to the problem of asynchronous VLSI design. Trace Theory is a mathematical framework that describes sequences of symbols from an alphabet , called *trace sets*, that specify the required behaviour of a circuit. Ebergen [Ebe87] used trace theory to design speed-independent circuits. Dill used trace theory to specify the required behaviour of a circuit or system [Dil89], and then showed how to *verify* (i.e. prove) that a given circuit meets the specification.

Tangram

Tangram is a programming language using extended CSP constructs [Ber92b] that is compiled into an intermediate representation called *Handshake Circuits*. Handshake circuits are then compiled directly to silicon, however, many optimisations and system requirements (e.g. initialisation and testability needs) can be added at the handshake level before the final compilation stage. The Tangram language and its associated decomposition processes are well-integrated in a tool-set, and are being used in an industrial context.

Occam translation

Brunvand [Bru91] described a synthesis procedure that converted a subset of the OCCAM language into circuits. The approach allowed conversion into either two or four phase circuits, although the former was chosen for the purposes of the study. Occam statements were decomposed into *constructs* for each possible program statement, and the resultant structure was then optimised using a variety of techniques before a final circuit structure was produced.

Other Methods

A close alternative to using CSP-like descriptions is CCS (*Calculus of Communicating Systems*) [Mil65]. However, this method has not been widely adopted due to the popularity of CSP-like languages.

Josephs describes SI/DI-algebra [JU90b, JU90a, JU93a, JU93b], a generalised method describing hardware systems. Unfortunately, the notation for circuits becomes cumbersome very quickly as the size of the system increases.

Endecott has developed LARD [End97a, End97b], a powerful software language that borrows features from Tangram, CSP, and VHDL for a unique asynchronous hardware description language. The language is, however, not generally suitable for direct silicon compilation, and is more geared to the specification and simulation of an asynchronous design at a high level of abstraction.

1.3.3 Engineered Approaches

Perhaps the best known example of an engineered approach is Sutherland's *micropipeline* [Sut89], shown in Figure 2.9. The micropipeline uses two-phase signalling and special *capture-pass* latches. The *capture* (or *cap*) signal latches data (i.e. closes the latch), and *capture-done* is issued by the latch when this action is complete (*captD* in Figure 2.9). The *pass* signal opens the latch, with *pass-done* being issued when the latch is open (*passD* in Figure 2.9). The inversions on the C-gate input from the subsequent stage *capture-done* signal allow the pipeline to operate correctly for the first operand.

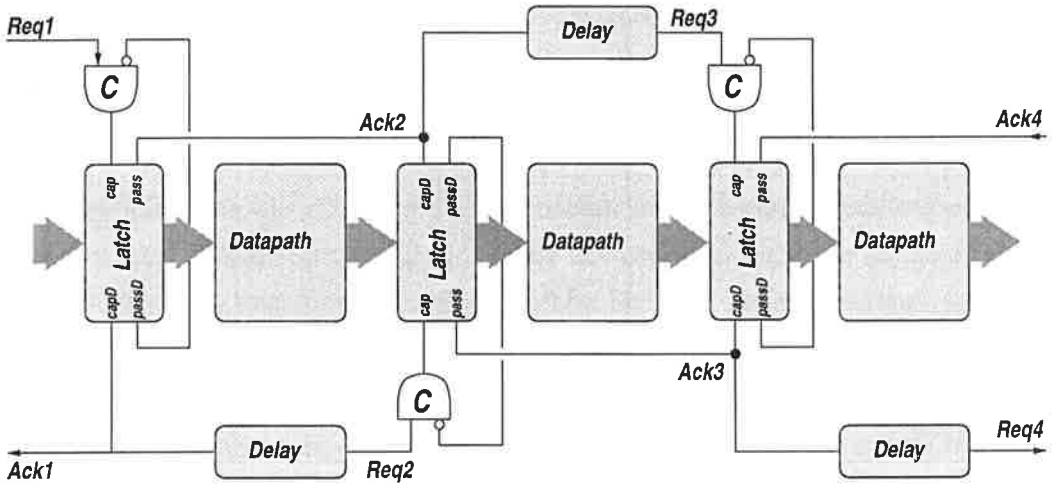


Figure 2.9 The Micropipeline Structure. The latches are special capture-pass event registers, but the datapath is *bundled-data* and can be designed using a variety of techniques.

Although the micropipeline is conceptually elegant, in practice the *capture-pass* latches are very large and expensive when implemented directly. Sutherland suggested a circuit based on the *toggle* gate in order to use level-sensitive latches, as well as a group of logic elements for implementing general control functions.

Although not a complete approach in itself, Sutherland's contribution was critical because it showed that the asynchronous control path could be decoupled from the datapath, allowing the datapath to leverage traditional design practices instead of more difficult and expensive fully-asynchronous methods. The asynchronous control could then be implemented in a number of ways, either by a two-phase SI method [Sut89], or a four-phase method [DW95, FD96], and could use delay elements and SR computation instead of more expensive DR computation. The success of this approach is evidenced by the volume of work that has built upon it [FDG+94, Pav94, Røi94a, SSM94, KdSRA91, LM92, Kel95, DW95, Bru91, Lip94, Røi94b, CL95, GO95].

Other Methods

Although early work on engineered methods focused on fundamental mode operation [Huf64], which severely limited cycle time, later methods broke fundamental mode requirements in some manner. Burst-mode circuits [Now93] react to unique input signal sequences, *firing* a number of outputs when the input sequence has been recognised. These circuits typically use a local clock (which helps in avoidance of some logic hazards), but there is no global skew requirement between interacting burst-mode circuits. An extension of the burst-mode approach, *3D* [YD92, Yun94], does not require a local clock, and can be applied to fully speed-independent designs as well as bounded-delay circuits.

An interesting concept is that of *speculative completion* [Now96, NYB97]. Speculative completion uses a group of *matched delays* to mirror the various speeds of the datapath, selects the smallest delay *speculatively*, and then aborts to other (higher) delay values if the data proves not to match the speculation. This approach avoids the area overhead of full dual-rail computation and the associated time wasted in the completion circuit, while retaining most of the time advantage of full asynchronous operation. The abort network must produce valid abort signals by the time the shortest path time has elapsed, and thus introduces timing constraints into the datapath (as would be expected, as this is a single-rail approach). Using 32-bit Brent-Kung adders operating on random data, the authors report a 19% performance improvement over a synchronous implementation using the same carry scheme [NYB97].

Pucknell [Puc90, Puc93] used a method for describing two-phase asynchronous circuits called *transition equations*. Transition equations describe the functionality of a circuit or gate in terms of transitions or non-transitions on the input signals, for example an AND gate would be described as,

$$\begin{aligned} \triangle Out &= \triangle a \cdot \overline{\bigtriangledown b} + \triangle b \cdot \overline{\bigtriangledown a} + \triangle a \cdot \triangle b \\ \bigtriangledown Out &= \bigtriangledown a + \bigtriangledown b \end{aligned}$$

The suitability of transition equations to describing two-phase systems will be briefly discussed in Chapter 3.

1.4 Hardware

In contrast to the number of design styles, very few actual working asynchronous hardware systems have been demonstrated.

The first asynchronous microprocessor was developed by Martin [MBL+89a, MBL+89b] using the CHP approach. It is was a 16-bit device with a three-stage fetch-decode-execute pipeline. The complete processor was specified in CHP, and compiled to layout using a combination of manual (for the datapath) and automated (for the control logic) techniques. An implementation in 1.6μm CMOS achieved 18MIPS at room temperature [MBL+89b], and the designers describe several measurements performed that show the robustness of the device, achieving 30MIPS at 77^oK.

The design was re-targeted for GaAs implementation [TMBL94, TMBL93] using DCFL [LB90]. The designers used a special structure for a NAND gate, which may not have been advisable considering the arguments for not using such a structure in DCFL GaAs logic [BS95a] for noise and propagation delay reasons. The decomposition process used for the original device [Mar90a], tuned to create trees for CMOS logic, had to be modified to mesh with the

low fan-in and fan-out requirements of GaAs technology. The second implemented device achieved 100MIPS, a performance drop of 50% from simulated values. The designers suspected CAD tool problems and parasitic under-extraction as possible causes, although the full functionality of both the initial and the re-designed parts is testament to the robustness of the QDI approach.

This group also produced a highly-pipelined digital lattice filter [CLM94]. The computations were bit-serialised, which proved to be an interesting approach as every bit can then be locally timed. However, the size and power consumption of the fabricated device was excessive — with 266 000 transistors in an area of $34mm^2$, it consumes 7.33W at an equivalent rate of 184MHz to do two 12-bit multiplies and two 12-bit adds per "cycle". However, the design achieved an equivalent clock rate of close to 200MHz, which would have been difficult to achieve with a synchronous implementation in the chosen process (0.8μm CMOS), even if a bit-serial architecture were employed.

Nanya described a speed-independent 8-bit accumulator-based microprocessor [NUK+94] implemented in 1.0μm CMOS SOG technology. The design achieved 11.2 MIPS at a supply voltage of 5V. The authors identified the delay model as being cause for some performance and area problems, saying of the (speed independent) delay model

> *... is not necessary simply to guarantee correct operations of the CMOS gate array microprocessor in a practical environment.*

Komori described a self-timed pipeline scheme [KTT+88] using modified Muller-C elements for synchronisation, reporting a maximum throughput of 18ns in a ≈1.5μm CMOS process. This technique was applied in the design of a floating-point unit [KTT+89].

The AMULET project at the University of Manchester has produced two asynchronous microprocessors based on the commercial ARM architecture, AMULET-1 [Pav94] and AMULET-2 [FGT+97]. AMULET-1 used a micropipeline approach, which the designers found was a bottleneck in performance due to communication overheads caused by the signalling style [DW95]. AMULET-2, their second-generation processor, used a four-phase communication approach, with appropriate pipeline controllers, which was more efficient [FD96]. The control structures used in both devices are speed-independent, but the datapath is bundled-data bounded-delay to improve area and time efficiency.

AMULET-1 was functional in both 1μm and 0.7μm CMOS over a temperature range of -50^oC to 120^oC, and operates down to 2.5V, which in itself is interesting because the majority of the system used delay models to mirror datapath completion times. The 1μm part delivered 20.5kDhrystones under nominal conditions. The project also demonstrated the feasibility of

re-targeting a synchronous instruction set to the asynchronous domain. AMULET-2 improved on the AMULET-1 by incorporating a number of architectural improvements. The design incorporates on-chip RAM, branch prediction, and result bypassing. The AMULET-2 also includes features necessary to make it usable in an embedded environment — interrupt control and timers [AMU96].

The Tangram approach developed at Phillips [Ber92b] has been used to design a DCC (digital compact cassette) error corrector system [BBK+94a, BBK+94b, BBK+94c]. The system also incorporated a novel power and speed management technique [NK94, BBK+95] which monitored the status of an input asynchronous FIFO, and adjusted the supply voltage accordingly. The authors reported dramatic power savings (80%) over an equivalent synchronous implementation with a modest area penalty (70-100%). The design was subsequently re-targeted to use single-rail handshaking and datapath structures [BBK+95, Pee96], reducing the area overhead to 20% of that of an equivalent synchronous implementation.

Williams described a fast pipelined 54-b divider [WH91] using four-phase SI communication with novel techniques for improving throughput without sacrificing latency. The general framework of trading latency for throughput was also described [Wil90, Wil91]. Meng [Men88] describes the implementation and testing of some asynchronous DSP structures, reporting a peak multiplication speed of 62.5ns for 16-bit operands in 1.6μm CMOS.

McAuley used a one-hot-encoded scheme to transmit data in an asynchronous FIFO at a high rate [McA92b, McA92a]. Instead of encoding three states (TRUE, FALSE, NULL) on two wires, four wires are used to encode two states (TRUE, FALSE) switching between two phases which must alternate when data is transmitted. A change in phase indicates that data has arrived, and can be acknowledged when required. A FIFO fabricated in 2μm CMOS ran at 500MHz.

Yun describes a differential equation solver [YDA+97], which incorporates an asynchronous adder modified to use carry-skip [WE95] techniques and asymmetric completion detection trees to reduce the completion detection overhead when the input operands result in longer carry chains. This adder design gives significant worst-case speedups over existing asynchronous adders. The design was fabricated and solved the target equation in an average of 35.6ns, and also used timed-circuit verification tools [XB97].

Morton described a multi-chip FFT processor [MAL94], which used the ECS approach and dynamic pipelining techniques to achieve high throughput. A simplified implementation of the system has been fabricated but not tested.

2 Related Work

Chapters 7 and 8 share similar objectives with two existing solutions to the same perceived problems. STARI is a solution to the inter-chip bandwidth problem, and STRiP is an adaptation of asynchronous concepts to a semi-synchronous method.

2.1 STARI

Greenstreet [Gre93] described a semi-synchronous chip-to-chip communication mechanism called STARI, *Self-Timed at Receiver's Input*. An overview of the STARI system is shown in Figure 2.10

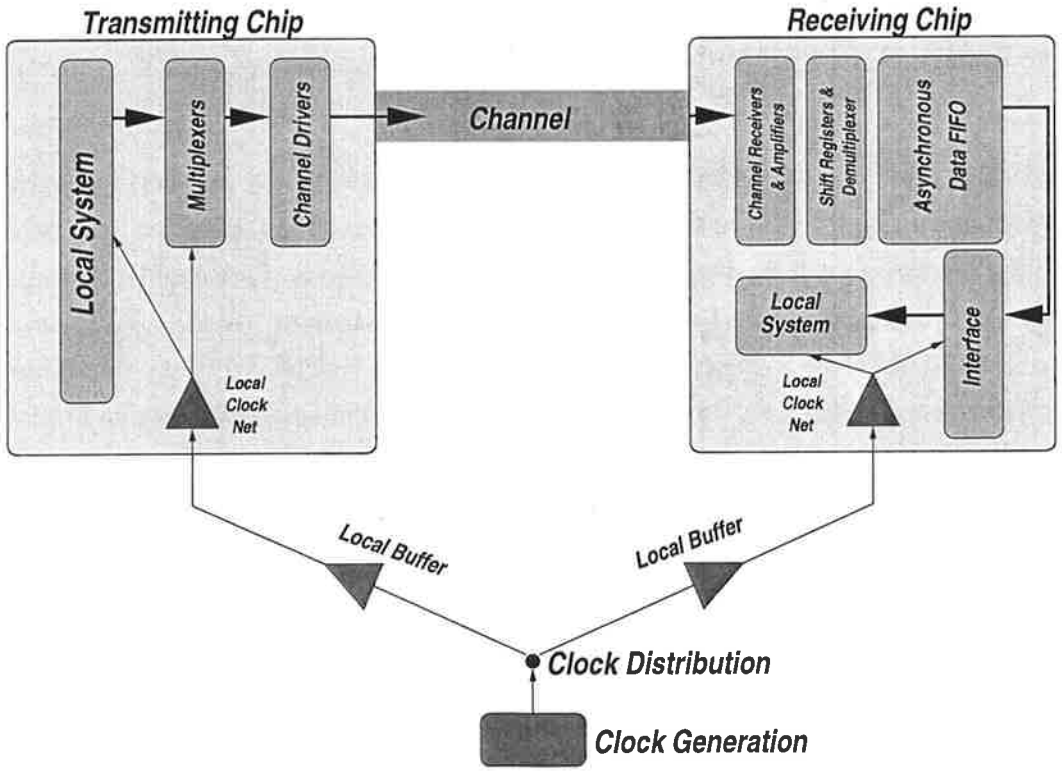


Figure 2.10 STARI system overview. The communication system is highly skew tolerant as the receiving interface is totally asynchronous.

STARI is designed to eliminate the chip-to-chip skew problem that makes high-bandwidth communication difficult between synchronised devices. The Transmitting system sends data to a multiplexer and driver circuit, which encodes the clock and data and sends the data over a dual-rail bit-serial channel. The receiving system is totally self-timed (and thus is highly tolerant of any data skew, in fact, the self-timed receiver is completely insensitive

to skew because it does not interact with the clock), and places incoming data into a shift register to realign it to the required word size. The asynchronous FIFO receives data words out of the shift register. The FIFO's function is to compensate for mild skew variations between the two systems. The system requires careful attention to initialisation, as the FIFO is ideally initialised to be half-full (to maximise its ability to absorb skew variations, and so that there are no FIFO to local clock system synchronisation issues). The STARI system was proved correct using the hardware verification language *Synchronised Transitions* [Sta94]. The system was fabricated, but some circuit faults caused by analog circuit issues and timing races were found, although the system was partially functional at 50MHz in 2μm CMOS.

In Chapter 7 a fully-asynchronous signalling system is developed that bears some similarity as it is highly skew-tolerant. However, the scheme is for communication between asynchronous systems, and as such its structure and operation is very different from the synchronous model used in STARI.

2.2 STRiP

Dean [Dea92] observed that a considerable benefit can be obtained by varying the clock rate of a synchronous processor in response to variations in functional unit latencies, and developed the STRiP architecture to take advantage of this. Asynchronous pipelines can transparently change cycle time in response to functional unit latency changes (caused by different logic depths seen by different instructions in the same stage), but STRiP does this within the synchronous framework. An overview of the linear STRiP processor is shown in Figure 2.11.

The processor pipeline is a standard early RISC design, and was adapted from a previous project [Cho89]. The novel feature is the *Dynamic Clock Generator*. Tracking cells in the generator, selected by information on the present worst-case operation in the pipeline being processed, mirror the delay of their respective functional unit, and this timing information selects the length of the next clock period (the timing information is selected one cycle in advance). This data is sent to a clock generation system, where the resultant two-phase global clock is distributed to the pipeline. The clock pulsing method creates a set clock high time (the $\phi 1$ clock phase), and a clock low time (the $\phi 2$ clock phase) that varies with the tracking cell selected by pipeline feedback, and thus the circuit blocks cannot depend on a set duty-cycle in the clock (which may require dynamic blocks in some systems to use specially-designed circuit structures, such as self-timed precharge).

The STRiP approach is interesting because it shows that up to 100% performance improvement may be gained simply by taking advantage of coarse-grain variations in functional unit latencies, for example the latency difference in an ALU between an ADD and a BRANCH operation. Fine-grain latency variations, such as latency changes caused by data-dependent

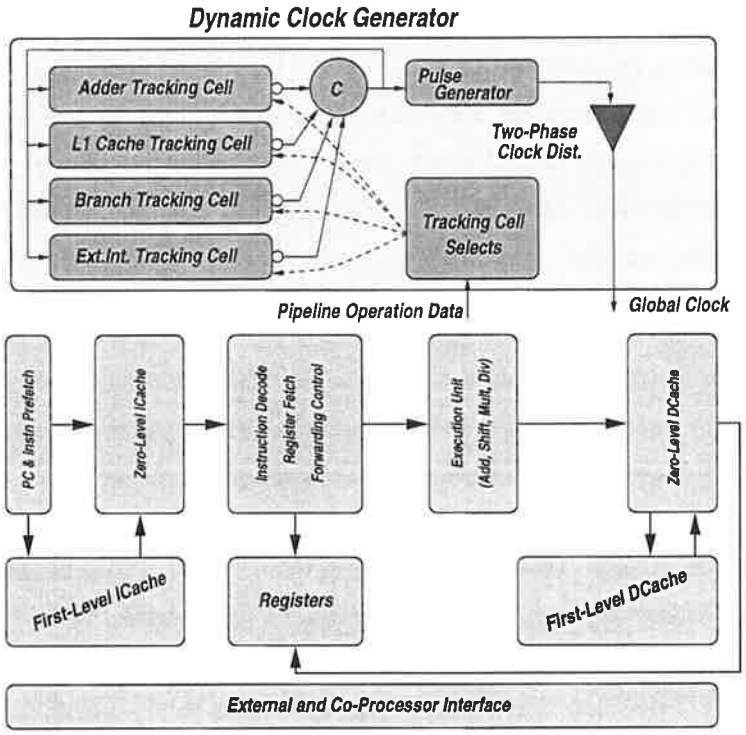


Figure 2.11 The STRiP processor. Tracking units vary the clock period as the latency of each of the functional units changes in response to changes in the instructions being processed.

operations (for example, the self-timed adder [Gar93]), are only exploitable by asynchronous systems. The dynamic clocking approach keeps the synchronous framework in place, thus removing the timing overheads caused by asynchronous operation (an explicit goal of the STRiP approach). Unfortunately, STRiP was not fabricated, and the performance advantage could not be quantified in practice.

3 Conclusion

It is apparent that the majority of asynchronous design methods employ a speed-independent delay model, and are concerned with producing correct-by-construction circuits without requiring that the designer consider technology-dependent factors. Even though one of the stated goals of QDI/SI methods is removing implementation concerns from the designer, Hauck observes of the QDI method [Hau95],

> *it is important to realise that many of these steps require subtle choices that may have significant impact on circuit area and delay ... much of the effort is directed towards aiding a skilled designer ...*

Thus, in order to produce a *good* circuit, the designer must have an understanding of not only the decomposition process, but to some extent the target technology and what effects it has on the results of the decomposition. Even armed with this knowledge, the performance (area and time) of SI circuits is still much lower than their synchronous counterparts in the same technology [BBK+95]. However, SI techniques do produce systems which are highly tolerant to parameter variation (process, temperature and voltage) and have no global signal distribution problems (i.e. clock signals).

The obvious choice is then to explore a method in which timing assumptions are made in order to improve the area and time performance of asynchronous circuits, analogous to the synchronous model (in which timing assumptions must be made — all computation must be completed by the end of the clock cycle). Methods using bounded-delay models can be divided into two main classes. One class is essentially a retrofit of SI synthesis techniques, with timing assumptions allowing timing-redundant checks and guards to be eliminated [Mye95]. The other is the *engineered* group of techniques, such as micropipelines and the methods used by the AMULET project. These techniques use speed-independent controllers, generally use single-rail logic techniques, and employ bundled-completion for the datapath. However, using speed-independent control in critical parts of the system still causes a large performance degradation.

Pseudo-speed independent techniques may eliminate some of the performance overhead caused by using the most general delay model, but the user will still be *locked in* to the method of implementation preferred by the method (e.g. language decomposition or state enumeration of a graph). It is clear from current design practice that synchronous techniques can obtain the performance levels they do because the designers have total control over all implementation styles and gates, even the choice of latch designs [BBB+95]. This *knowledge-based* model is widely successful, and is supported by a large base of algorithms and techniques for timing analysis of very large digital systems.

3.1 Let's go faster ...

Even now, there is still a very great need to improve the achievable performance of asynchronous methods. Some of the most efficient 4ϕ controllers still have cycle times (when operating as FIFOs) that are comparable to the *complete* cycle time of a synchronous system implementation in the same technology [FD96]. This is clearly makes asynchronous design undesirable from a practical viewpoint.

The approach described in this thesis is to use an approach similar to the synchronous method, giving the designer total freedom and control over the implementations of everything from gates to systems. Although this method is the most expensive in terms of designer effort, it

will produce the best results when used by a skilled designer.

Therefore, we move to a bounded-delay model in *all* parts of the system, including the control components. This will require timing constraint definition and checking, which is the penalty we choose to pay for extracting higher performance from a given technology. We also choose to use two-phase signalling. Four-phase (4ϕ) signalling has been shown to be more efficient than two-phase (2ϕ) in a speed-independent environment [DW95]. Signalling using 2ϕ has often observed to be potentially more efficient because it involves fewer signal transitions on *req* and *ack* wires, but this has not been demonstrated in practice, despite the huge popularity of the *micropipeline* approach (in Chapter 4.1, the potential speed advantage of 2ϕ pipelines over 4ϕ pipelines shall be demonstrated, establishing a significant performance gain for 2ϕ signalling approaches).

Throughout this thesis, the objective will always be to maximise the speed performance of the circuits and systems under investigation. This is done often at the expense of robustness to timing and parameter variation. Often, improving the performance of asynchronous systems involves moving towards more timing verification, driven by well-defined timing constraints. The methods used in this thesis can be classified as 2ϕ signalling bounded-delay systems, generally using single-rail datapath circuits with bundled-data completion.

Chapter 3

The Event Controlled Systems Design Methodology

EVENT CONTROLLED SYSTEMS is a methodology for the design of two-phase asynchronous systems. This approach is, in general, a bounded-delay modelled, single-rail, bundled-data asynchronous design method.

ECS *is not* a framework intended for synthesis or formal verification in its present form, nor is it a speed-independent approach. ECS *is* designed to provide tight control over the low-level implementations of a system and the decomposition process employed, which was stated in Chapter 2 as a requisite for higher performance.

In this chapter, a new notation is developed that exposes the simplicity of two-phase signalling at the gate level. Previous two-phase methodologies have not attempted to provide such an intuitive understanding to the operation of two-phase networks. This notation exposes a group of very simple, *atomic* gates used for the design of control circuits. Many of these gates are *not* speed-independent, and are used in control networks which require timing verification. The system-level issues of error detection and simulation encountered when interconnecting these gates are then described. Finally, the ECS notation is modified to move it towards a more semantically complete form, and a graph-based analysis tool using this new notation is described.

The concepts and techniques developed in this chapter are employed extensively throughout this work as a basis for high-speed asynchronous design, and ECS is the methodology used in the development of the ECSTAC microprocessor, to be presented in Chapter 5. The ECS approach was developed by Morton [Mor97], but the author made contributions to its development.

1 Two-Phase Design

Two-phase asynchronous approaches have not received as much interest as four-phase approaches, perhaps due to the problems in providing a formal basis to the approach. Two-phase methods have typically used a modular block-based approach [Sut89], successfully applied in design experience [WDF+97].

However, this block-based approach is unsuitable to a well-tuned design approach because it may not expose the lowest levels of implementation for optimisation and can thus be restrictive. The design process also tends to be *ad-hoc*.

1.1 Previous Two-Phase Signalling Formalisms

Some past work has attempted to apply CSP-like constructs to two-phase systems, typically to a group of operators that facilitate modular composition. Ebergen [ESB95] uses prefix commands to model systems that are composed of two-phase operators, such as TOGGLE and JOIN, and produces speed-independent versions of modulo-N counters as an example of the decomposition process. Such an approach is not particularly suitable for high speed implementations of systems, since it offers no direct control over the actual circuits.

1.1.1 Transition Equations

Pucknell's *transition equations* [Puc93, Puc90] are another method of description that is applicable to both boolean and event gates and signals. A distinction is made between *transitions* and *non-transitions*, i.e.

$$\begin{aligned} \triangle A &\Rightarrow \text{signal A goes to logic } \mathbf{1} \\ \overline{\triangle A} &\Rightarrow \text{signal A stays at logic } \mathbf{1} \end{aligned}$$

This distinction can make the representation of simple gates somewhat complex. For example

$$\begin{aligned} \triangle X &= \triangle A \cdot \triangle B + \triangle A \cdot \overline{\triangledown B} + \overline{\triangledown A} \cdot \triangle B \\ \triangledown X &= \triangledown A \cdot \triangledown B + \triangledown A \cdot \overline{\triangle B} + \overline{\triangle A} \cdot \triangledown B \end{aligned}$$

describes the Muller-C gate, which is intuitively an *AND* gate for transition signalling. However, the notation does not explicitly show this correspondence.

Transitional operators can be grouped to give a notation for *events* and *non-events*,

$$\begin{aligned} \partial A &\Leftrightarrow \triangle A + \triangledown A, \text{and} \\ \overline{\partial A} &\Leftrightarrow \overline{\triangle A} + \overline{\triangledown A} \end{aligned}$$

and thus the ∂A operator indicates the occurrence of an *event*, or *transition*, on the line A. The event operator is an important concept, since it defines the occurrence of a transition as being independent of the logical state of the signal. The ∂A notation shall continue to be used for events on signal lines. However, the transition-equation representation tends to get cumbersome because gate equations must be written in terms of events *and* non-events on signals.

It is apparent that an even simpler representation is required that enables ready formalisation of two-phase systems. This formalisation should also be compatible with traditional mechanisms for describing boolean-type gates without excessively verbose equations. The method by which this can be achieved can be illustrated by examining a simple two-phase handshake.

1.2 Two-Phase Asynchronous Signalling

Signals which are viewed as two-phase do not have truly *boolean* values. Examining only the logical state of an isolated signal makes it impossible to tell what *state* the signal is in from its boolean value. However, being able to determine the *status* of a two-phase signal would be advantageous because it would bring structure to the specification of two-phase systems.

To determine whether such a structure can be found, a two-phase handshake, shown in Figure 3.1, is examined. $\partial Start$ events are indicating that some action be taken, while $\partial Done$ events are indicating that this action has been completed, or at least started.

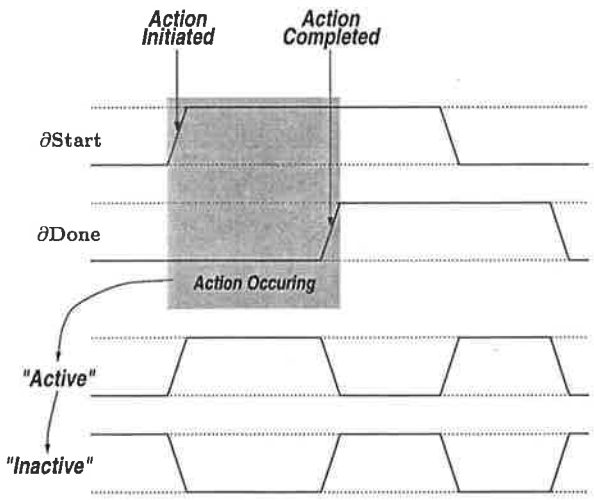


Figure 3.1 Two-Phase Communication Sequencing.

If the system is truly asynchronous, a new $\partial Start$ event should not be issued until the circuit sends the $\partial Done$ event. More concisely, a $\partial Start$ cannot occur while the circuit is still in

the *active* state of Figure 3.1. A two-phase notation which explicitly described the state of a signal as *active* and *inactive* would be preferable to a boolean description, since a boolean description of a system using transitions as a communication mechanism is ineffectual.

1.3 Transformational Approach

To expose the required *status* information desired in a two-phase representation, all signals are *transformed* to a new domain, the *temporal domain*[1]. The *temporal transform* converts any signal, event or boolean, to this new domain, the *temporal domain*. The transform is transparent for normal boolean signals (named *data lines*),

$$\begin{aligned} \triangle A_l &\Leftrightarrow \triangle A_t \\ \triangledown A_l &\Leftrightarrow \triangledown A_t \end{aligned}$$

where the subscripts l and t refer to the logical[2] (the actual physical domain of 0s and 1s) and temporal domains respectively. As the *data* signals already carry boolean information, they are unchanged through this transform, thus the logical value of a signal is identical to its temporal value. The transform applies to event lines in a different way. The occurrence of the event, as was previously discussed, indicates some *action* is required, and thus an event transition causes an *up* transition on the transformed representation of the signal,

$$\triangle \partial A_l \; + \; \triangledown \partial A_l \Rightarrow \triangle \partial A_t$$

This temporal domain signal ∂A_t is indicating the *active* status of the signal (like that of Figure 3.1). Note that the ∂ notation is used to signify two-phase *event* signals. This now frees up the negative transition of this transformed signal, as *any* transition on the logical signal results in an $\triangle$ transition in the temporal domain. The reverse transform, going from the temporal domain to the logical domain, is more complex, as the initial state of the signal must be known. If there is a $\triangle$ transition on a temporal event signal at time x, then the reverse transform is

$$\triangle \partial A_t|_{t=x} \;\; \Rightarrow \;\; \partial A_l|_{t=x} \leftarrow \overline{\partial A_l|_{t=x-\epsilon}}$$

A $\triangle$ transition on ∂A in the temporal domain at $t = x$ causes a transition on the signal in the logical domain, given by the inverse of the signal just before the transition occured (where ϵ is some small number). The negative transition on temporal event lines can now be redefined. As was suggested by Figure 3.1, the *effect* of an event has been processed when the corresponding *output* event occurs, and this is the definition chosen for the $\triangledown$ transition on a temporal event line. However, for two-phase gates the definition of the *output event* varies according to the gate function.

It is worth mentioning at this point that physical implementations of temporal signals are never tractable, as the concept is *not physical*. Temporal representations of signals are useful for checking system functionality, as well as providing a framework for the understanding of two-phase control and its interaction with normal boolean variables. In implementation, the logical form is the implemented system, as would be expected!

1.4 Operators and Equations

Now that the ability to specify two phase signals has been established, the way in which this is used to specify gate functionality is described. A number of operators can be used to describe the operation of two-phase gates. The most important temporal operator is that of *assignment*, '$\leftarrow$'. The general form of the assignment operator is

$$signal \quad \leftarrow \quad (temporal - variable - expression)$$

The value of *signal* is assigned to the evaluated value of the expression. An assignment is present in any *temporal equation* (TE), which describes the behaviour of a signal in the temporal domain, and can be generally be viewed as *effect* $\leftarrow$ *cause*. The mechanism by which any event signals in ($temporal-variable-expression$) are set low will be discussed in Section 1.5. From this point on, any use of an event (for example, ∂in) refers to its *temporal domain* value unless otherwise noted.

Logical — AND and OR

Logical operators can be used between two or more temporal values. For instance, two *event* lines can be ANDed or ORed,

$$\begin{aligned} \partial C &\leftarrow \partial A \cdot \partial B \\ \partial F &\leftarrow \partial C + \partial D \end{aligned}$$

Boolean lines can also be ANDed and ORed, and any temporal variable can be ANDed with any other. However, the following expression is not valid,

$$\partial O \leftarrow \partial A + B$$

as if B becomes true, there is no way to generate $\bigtriangledown \partial O_t$(see Section 1.5) which is necessary to construct a gate. Other temporal equations (TEs) can be envisaged which also fall into this illegal class [Mor97].

UNTIL — U

The *Until* operator acts exactly as its name suggests — it evaluates true after some condition comes true *until* another condition becomes true. This can be used for *data* variables or *event*

variables, for example

$$A \ \mathbf{U} \ B$$
$$\partial set \ \mathbf{U} \ \partial reset$$

Normally the two variables would be mutually exclusive (for instance, with data values $A + B = 0$ always), but this is not required. The response of the circuit or gate to such a condition is implementation dependent and left up to the designer's discretion. Note that the until operator applied to events *enforces* exclusivity. As the result of the *until* operator is always a *boolean* signal, this cannot serve as an output event for the input events which set the signal. Thus, the ∂set and $\partial reset$ events function as output events for one another, enforcing exclusivity (see Section 1.5).

Conditional — > (or *after*)

The *Conditional* operator allows the evaluation of the expression in a TE assignment to be conditional upon another temporal expression. The general form is

$$signal \ \leftarrow \ (expression) > (conditional)$$

The conditional operator operates to *annul* any assignments to *signal* that result in the (*expression*) changing its state. The exact use of the conditional will be illustrated by the *Feed*, *Restore* and *Latch* gates of Section 2.

Depends Upon — :

The *Depends Upon* operator allows dependencies to be made explicit in a temporal equation. For instance, if an event $\partial later$ is not to be generated from $\partial cause$ until the signal a has gone *valid* (i.e. the signal has been *set* appropriately, and is not necessarily $a = 1$),

$$\partial later \leftarrow a \ : \ \partial cause$$

This operator is only used to make dependencies between signals *explicit*, and in implementation usually corresponds to a delay operator.

1.5 Gate Operation

The $\bigtriangledown$ transition of any *event* signal has been opened for redefinition in the temporal domain. This $\bigtriangledown$ transition is caused by the $\bigtriangleup$ transition on the event signal which is considered as the *output event* of this signal. In a temporal equation (TE), the enabling of the output event ($\bigtriangleup\partial out$) sets the input events low by this definition, eliminating the condition which caused the output event (since the enabling expression is now false), thereby setting it low. It follows that output events of TEs are only ever pulses (δ-pulses).

This mechanism is less restrictive than Martins' *Acknowledge Theorem* [Mar90b], which requires that any transitions which disable the guard of an expression be in the successor set of the guarded expression's output. In ECS, the gate *output* event resets the inputs *directly* — the acknowledge for a gate is not a successor of the gate output, it *is* the gate output. Interestingly, the reset mechanism for ECS gates occurs through temporal behaviour only, and the reset causes no *physical* change in the state of the circuit. This can be expected to provide a significant speed improvement simply due to the absence of an explicit physical acknowledge phase at the gate level, although this requires that the circuit be checked beforehand for errors (a defined procedure in ECS, described in Section 2.3), and the circuit will use a bounded-delay model in the general case.

Unfortunately, the output event which causes the $\bigtriangledown$ transition on input event lines in a general temporal equation (TE) is not necessarily the signal that is being assigned to by the '$\leftarrow$' operator, and is dependent on the operator being used in the TE, as will be illustrated in Section 2.

1.5.1 Operator Precedence in Gate Equations

Precedence relations between the various operators are shown in Table 3.1.

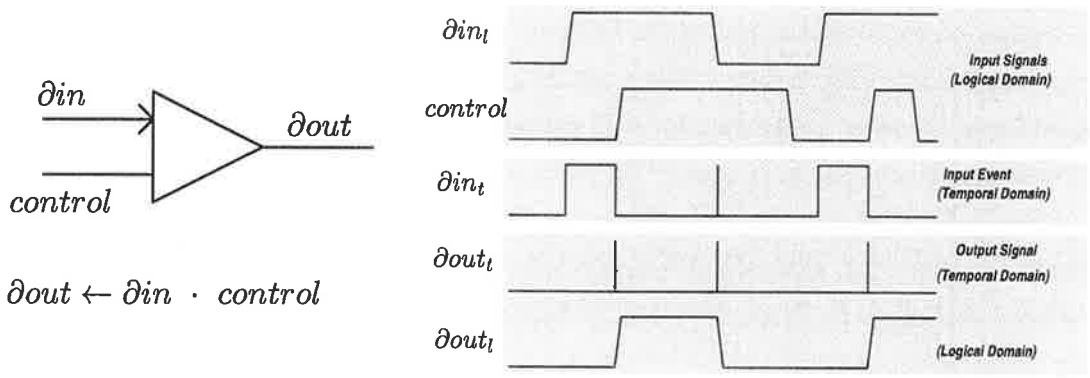
Precedence, highest to lowest $\Rightarrow$					
AND($\cdot$)	OR($+$)	Depends-Upon(:)	Until(U)	Assignment($\leftarrow$)	Conditional($>$)

Table 3.1 ECS Operator Precedence Relations. The highest precedence operator is the AND($\cdot$), with the lowest being the conditional ($>$).

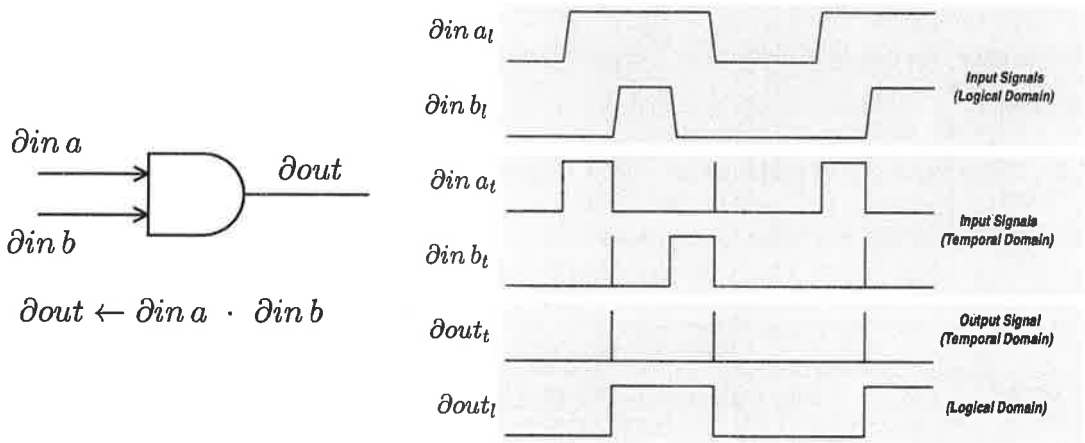
2 Two-Phase Gates

The fundamental two-phase gates based on this notation are shown in Figure 3.2. All gate equations contain the *assignment* operator, as this is an invariant part of any temporal equation (TE), and that they only contain *two* operators, including the assignment. TEs can be written with more than two operators, but such TEs are decomposable to two or more *atomic* TEs.

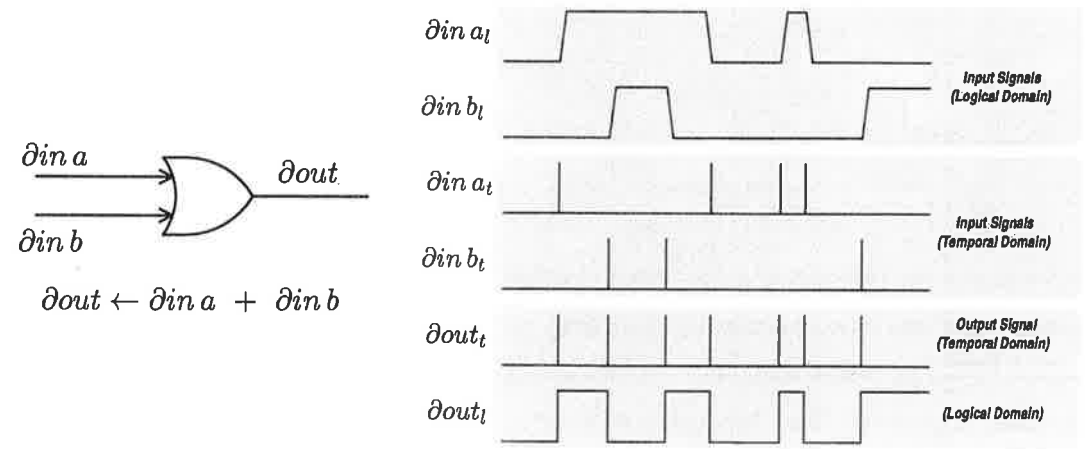
Unless otherwise noted, the signal on the left of the assignment (the *effect* part of the assignment) functions as the output event for any event signals on the right of the assignment (any event signals on the *cause* side of the assignment).



(a) *SEND* gate

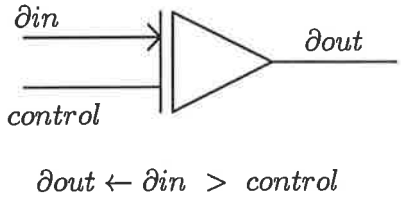


(b) *LAST* gate

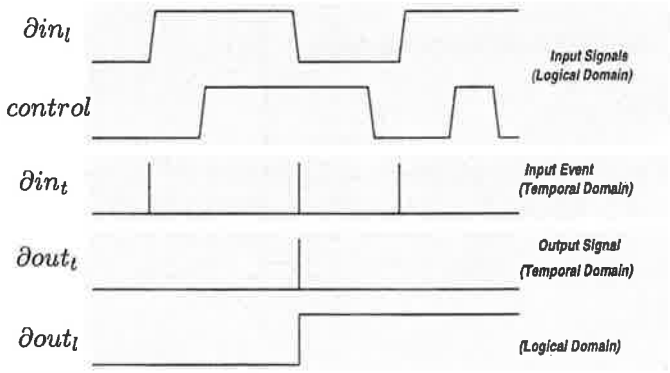


(c) *MERGE* gate

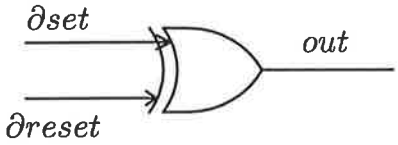
Figure 3.2 Fundamental ECS gates.



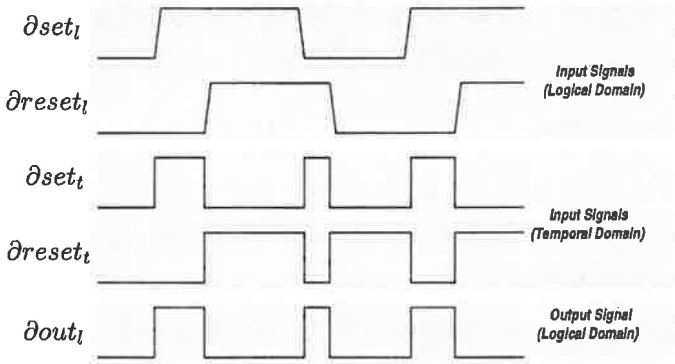
$\partial out \leftarrow \partial in > control$



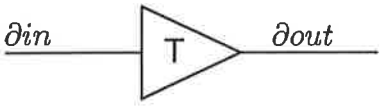
(d) *FEED* gate



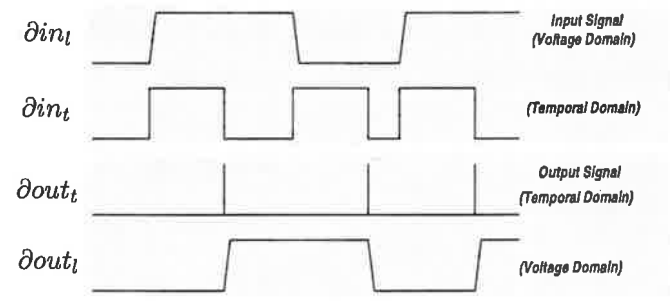
$out \leftarrow \partial set \; U \; \partial reset$



(e) *UNTIL* gate



$\partial out \leftarrow \partial in$



(f) *DELAY* gate

Figure 3.2 Fundamental ECS gates (Con't).

Send gate

The Send gate, shown in Figure 3.2(a), holds any events occurring on ∂in until the controlling boolean condition, *control*, becomes true. Any ∂in event arriving when *control* is already high passes straight through to ∂out.

Last gate

The Last gate of Figure 3.2(b) waits for both input events to become true, $\partial in\, a$ and $\partial in\, b$, before producing the output event, ∂out. It is equivalent to the commonly-used Muller-C or *Join* gate of other methodologies, but note that the ANDing function is now explicit in the TE.

Merge gate

The Merge gate of Figure 3.2(c) produces an output event, ∂out, after an event on either of it's input events, $\partial in\, a$ or $\partial in\, b$. The OR function that this gate intuitively implements on events is now explicit in the TE.

Feed gate

The Feed gate of Figure 3.2(d) only produces an output event, ∂out, when ∂in occurs *while* the boolean control signal, *control* is true. If *control* is not true, ∂in is reset and no output event occurs for this input. The feed gate must use ∂in as its own 'output' event (so that ∂in_t always appears as a pulse, see Figure 3.2(d)) because a ∂out does not always occur for every ∂in.

This gate can be used to construct larger two-phase gates found in other methodologies, for example the *Select* and *Toggle* of micropipelines [Sut89]. The *Select* gate is two *Feed* gates with complementary enabling conditions,

$$\begin{aligned} \partial select\, low &\leftarrow \partial in > \overline{select} \\ \partial select\, high &\leftarrow \partial in > select \end{aligned}$$

The *Toggle* gate is equivalent to the *Select* gate, however the *select* control signal is internally generated and toggles between two states. The specification for a *Toggle* is

$$\begin{aligned} \partial toggle\, first &\leftarrow \partial in > toggle \\ \partial toggle\, second &\leftarrow \partial in > \overline{toggle} \\ toggle &\leftarrow \underline{\partial toggle\, first}\ U\ \partial toggle\, second \end{aligned}$$

The generation of the *toggle* signal uses the *Until* gate.

Until gate

The Until gate of Figure 3.2(e) produces a boolean output, *out*, from two event inputs. The event which assigns the output *low* is drawn as the signal striking the outer arc of the circuit symbol, with the other event (assigning the output high) striking the inner arc of the symbol.

The definition for the output event of the until gate is unusual — as indicated in Figure 3.2(e), the input events to the gate in the temporal domain are mutually exclusive, and therefore the output event of an input to the until gate is defined as being the other input event. $\partial reset$ is the output event of ∂set, and vice versa. This is consistent with the definition of the temporal transform, as each input event has a unique event that resets its temporal truth.

Delay gate

The operation of the Delay gate of Figure 3.2(f) is straightforward, with the output event ∂out is produced T_d time units after the input event ∂in.

This illustrates one method of handling gate delays. When the conditions required to fire the gate are met, the input events remain true until the output event fires (after the appropriate gate delay time), and the input and output events are then reset normally (in infinitesimal time). In the context of ECS this is a hybrid inertial delay model. Morton [Mor97] uses a fully inertial delay model for the output of the assignment (where both the positive and negative transitions of temporal domain assignments are delayed by the gate delay) which is somewhat more restrictive as it delays the $\bigtriangledown\partial out_t$ transition as well, which is not strictly necessary. Another alternative would be use a *transport* delay model on the output of the assignment,

$$\begin{array}{lll} \partial out' & \leftarrow \partial in & \text{in zero time, } \partial out' \text{ resets } \partial in \\ \partial out & \overset{T_g}{\leftarrow} \partial out' & \text{i.e. delayed } T_g \end{array}$$

Using a transport-model for assignments is undesirable as it allows multiple input events to be incident upon the delay gate of Figure 3.2(f) before the output emerges. It also makes error detection complicated. The explicit definition of delays in two-phase gates is done by placing the delay above the assignment operator, e.g.

$$\partial out \overset{2.0}{\leftarrow} \partial in$$

defines ∂out as occurring 2.0 time units after ∂in.

Boolean gates

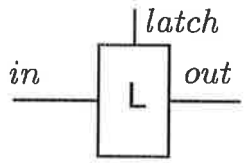
Boolean gates are still specified normally, for example

$$c \leftarrow a \cdot b$$
$$d \leftarrow c \oplus carry$$

The assignment operator is used here instead of the ' =' operator for consistency with other ECS gates.

Latch gate

The level-sensitive latch [SY92] is shown in Figure 3.3.



$out \leftarrow in > latch$

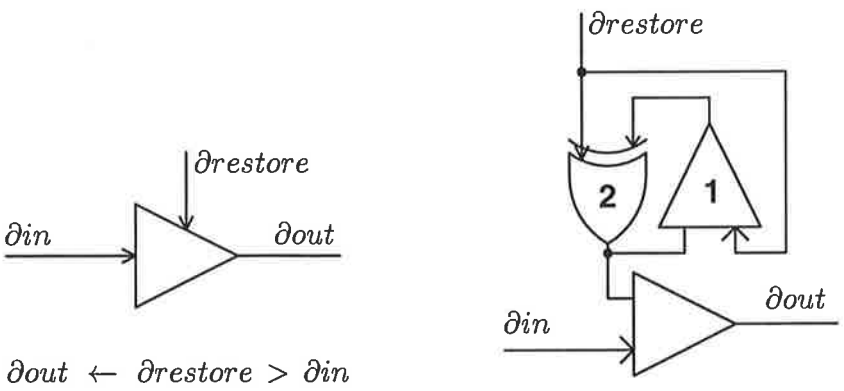
Figure 3.3 Level-Sensitive Latch in ECS.

Note that the use of the conditional assignment acts to annul any changes in the boolean signal *out* until the control signal, *control*, is high, and thus *out* retains its temporal value when *control* is low.

Restore Gate

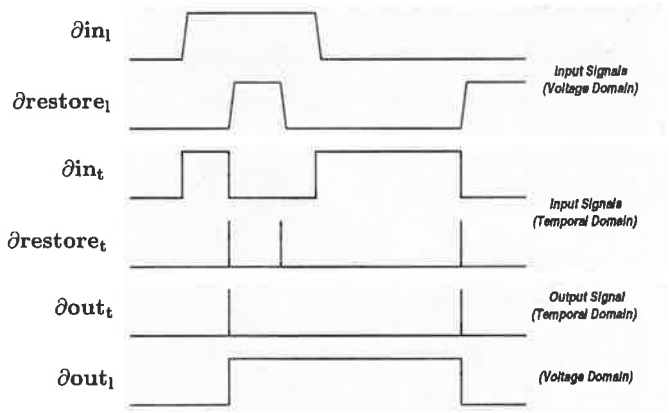
The *Restore* gate holds an input event, ∂in, at the input until the *restoring* event, $\partial restore$, occurs. When $\partial restore$ does occur, if no ∂in event is pending then there is no gate output event. The *Restore* gate is composed of a number of smaller gates in implementation, shown in Figure 3.4.

In this case, $\partial restore$ must function as the output event for the TE (similarly to the *feed* gate), as a $\partial restore$ event does not always result in a ∂out event. The Restore gate uses a pulse generator (consisting of gates 1 and 2 in Figure 3.4(b)), and thus a timing constraint exists on how the input event ∂in may arrive relative to the $\partial restore$ event. This constraint is equivalent to a *setup* time on the input of the gate, and the input cannot be changed until the enabling condition on the ∂in–∂out *send* gate has been reset.



(a) Symbol & Temporal Equation

(b) Implementation



(c) Behaviour

Figure 3.4 Restore gate. The Restore gate is an ECS gate that can be decomposed to a number of more basic gates, as shown in (b).

2.1 Initialisation

In general temporal descriptions, input signals to the gates will often be required to be viewed to be true (in the temporal domain) at $time = 0$, *initialising* the signal to a true temporal value. This initialisation specification is only applicable to *event signals* — if booleans are required to be initialised, then normal boolean logic equations must be used to make the initialisation explicit. Gate symbols make initialisation explicit by using a dot (•) on the relevant signal input to the gate. An initialised event signal, $\partial signal$, is written[3]

$$\underline{\partial signal}$$

For example, a *last* gate may be initialised to have an event initially true by specifying,

$$\partial out \leftarrow \partial req\, in \cdot \underline{\partial ack\, out}$$

which specifies the input *last* (or Muller-C) gate in micropipeline control [Sut89]. Initialisation is *required* of the *after* operator when using events to set the initial state of the output signal, for example

$$out \leftarrow \partial set\, U\, \underline{\partial reset}$$

sets the initial state of $out = 0$ (which would otherwise be ambiguous), as the $\partial reset$ event is specified as having occurred initially. This also requires that ∂set must be the next event to occur to avoid an *exclusion violation* (see Section 2.3).

2.2 Wire representation

The representation of wires in ECS is not trivial when considering event signals. The wire representation is shown in Figure 3.5, where the actual wire has the prefix ϕ attached to the signal name of its output gate (the wire is considered separately from the gate output).

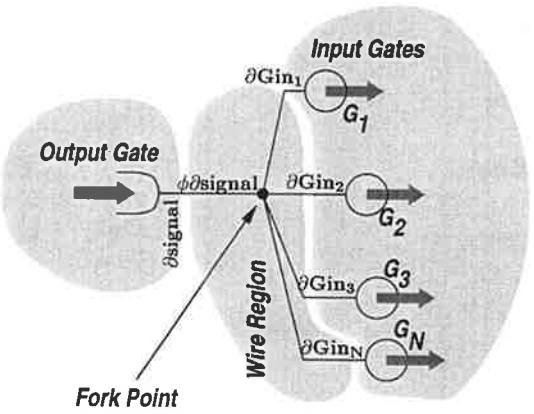


Figure 3.5 Event Wires in ECS. For boolean signals the wire is trivial, but for *event* signals we define the wire $\phi signal$ to allow for more advanced modelling.

Wires which distribute the output of a boolean gate to other gates are trivial, for example with a gate output signal *out*,

$$\begin{aligned} \triangle\phi out &\leftarrow \triangle out \\ \triangledown\phi out &\leftarrow \triangledown out \end{aligned}$$

and this ϕout signal is connected to all the gates using this output. However, for event wires the value of the output signal cannot be directly assigned to the value of the wire, since the output signal only appears as δ-pulses, and this would result in inputs being withdrawn from gates before their effects had been processed. Therefore, the wire must be maintained as

active (true in the temporal domain) until the effect of the output event has been completely processed. In this case, the events which constitute the set of gate inputs *using* this output need to be known, and the behaviour of the wire for an output event ∂out is described by

$$\begin{aligned} \triangle\phi\partial out &\leftarrow \triangle\partial out \\ \triangledown\phi\partial out &\leftarrow \prod_{i=1}^{N} \triangledown\partial Gin_i \end{aligned}$$

where G is the set of N gates using the $\phi\partial out$ wire as an input, with each gate input being labeled ∂Gin_i. The wire is only set low when all the inputs it feeds have gone low (in the temporal domain), where these inputs themselves have been set low by the action of the gate which they feed. Therefore, the gate inputs, gate outputs and signal wires, all of which have the same *logical* value, can have different temporal values at any one time. Figure 3.6 gives a summary of the mechanisms involved, which, although seeming overly verbose, make simulation and error detection in ECS systems almost trivial.

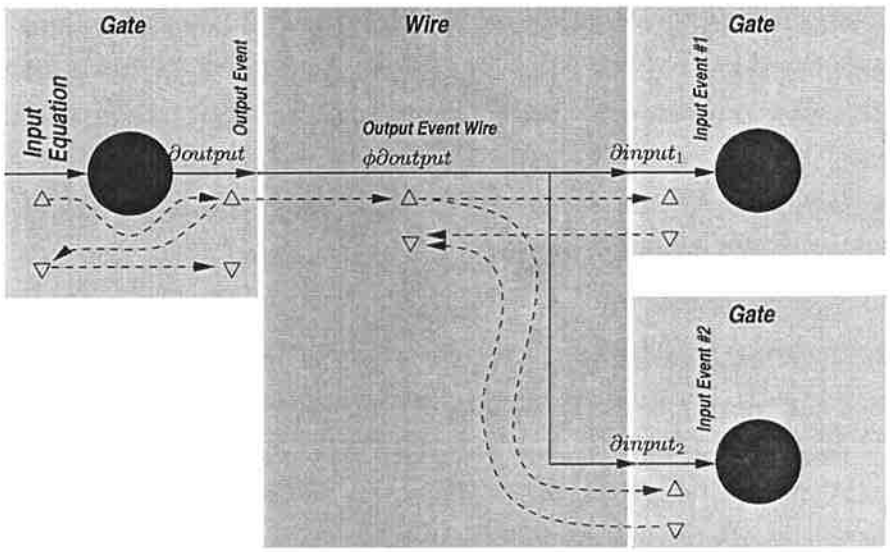


Figure 3.6 Causative Relations in ECS gate inputs, outputs and wires. The temporal values of gate inputs, outputs and wires can be different, even though they have the same *logical* value (if zero wire delay is assumed). This is because of the different mechanisms involved in the $\triangle$ and $\triangledown$ transitions for each component of the system.

2.3 System Simulation

The possibility of both simulating an ECS gate network and simultaneously detecting erroneous conditions in the network during simulation is now apparent. Gates and wires, which encode boolean information, can be described using traditional HDLs (such as VHDL

[LSU89]) and simulated, with both temporal and logical behaviours available for inspection. In addition, as the temporal domain signals are representing circuit activity or pending circuit activity at a particular point, the detection of two types of errors becomes possible. These two errors are known as *exclusion* and *coincidence* errors.

Exclusion Errors

An *exclusion violation* (EV) occurs when a signal transition occurs on an event line whose temporal value is already true, and is defined as

$$(\triangle \partial signal_l|_{t=x} + \bigtriangledown \partial signal_l|_{t=x}) \cdot (\partial signal_t|_{t-\epsilon} = 1)$$

any signal transition on an event line at any time whose temporal value is already true (ϵ is a small number, ensuring that we check just before the current signal transition) flags an exclusion violation. This error represents a *total* failure of the system being simulated, since it indicates that events are not being processed in a timely fashion before the next arrives.

Coincidence Errors

A *coincidence violation* (CV) occurs when a boolean signal enabling a gate goes low, or when certain changes in conditional evaluation (the $>$ operator), occur in close proximity to other input signal transitions which would otherwise evaluate the temporal equation of the gate to be true, thus enabling the gate. Such signal changes are undesirable since the output of the gate can become indeterminate, potentially causing serious errors in a transition-signalling system. Morton [Mor97] considers techniques for the detection of CV errors.

3 A Complete Syntax

The ECS notation as developed in Section 1 is very intuitive and compact, however, the notation does not handle output events consistently. Output events in general TEs are ill-defined, with the definition of output event generally depending on the operator(s) used in the equation, and it is not clear where the output events are unless the TE is broken up into constituent *atomic* TEs representing individual gates. We seek to improve upon the existing ECS representation by developing a slightly different notation called ECS′.

3.1 Improving Syntax

The output event of a given temporal equation (TE) is *defined* according to the operators present in the TE. In addition, in complex TEs it is not clear which event functions as the output event, if any, for all or parts of the expression, for example

$$out \leftarrow \partial start \cdot \overline{Done}\, U\, \partial done_a \cdot \partial done_b$$

Identifying the output events of each input event in the system will generally require a full expansion to its constituent atomic TEs, like those shown in Figure 3.2. In general, the *gate* output event is not the same as the event which is considered the *output* event of a gate, which can lead to confusion when attempting to make the distinction. For example,

$$\begin{aligned} \partial out &\leftarrow \partial in \cdot control \\ \partial out &\leftarrow \partial in > control \end{aligned}$$

In the first example, the *send* gate, the output event of the gate, ∂out, functions as the reset event. However, in the second example, the ∂in event functions as the reset event for the gate, although ∂out is the output event of the gate. Thus, the name *reset event* is used to define the event which *resets* gate input and outputs, and *output event* refers to the output event (if any) of the TE or gate under consideration.

In order that the definitions be made consistent, the reset event of a TE is made *explicit*, for example in the *last* gate,

$$\partial\hat{out} \leftarrow \partial in_a \cdot \partial in_b$$

in which $\partial\hat{out}$ is considered the reset event of the expression. This dramatically simplifies the equations and their behaviour, since it may be noted from the temporal behaviours of Figure 3.2 that most ECS gates behave either as AND or OR gates in the temporal domain, and it is the reset event that determines their behaviour. The resultant gate TEs are shown in Table 3.2.

Gate	Old TE	New TE	Figure Ref.
Send	$\partial out \leftarrow \partial in \cdot control$	$\partial\hat{out} \leftarrow \partial in \cdot control$	3.2(a)
Last	$\partial out \leftarrow \partial in\, a \cdot \partial in\, b$	$\partial\hat{out} \leftarrow \partial in\, a \cdot \partial in\, b$	3.2(b)
Merge	$\partial out \leftarrow \partial in\, a + \partial in\, b$	$\partial\hat{out} \leftarrow \partial in\, a + \partial in\, b$	3.2(c)
Feed	$\partial out \leftarrow \partial in > control$	$\partial out \leftarrow \partial\hat{in} \cdot control$	3.2(d)
Delay	$\partial out \leftarrow \partial in$	$\partial\hat{out} \leftarrow \partial in$	3.2(f)
Restore	$\partial out \leftarrow \partial restore > \partial in$	$\partial out \leftarrow \partial in \cdot \partial\hat{restore}$	3.4

Table 3.2 ECS′ Gate Representations.

Note that the $>$ operator has been eliminated, and that the inherent temporal behaviours of each type of gate are now apparent. In particular, both the *feed* and *restore* gates implement AND functions, and the new definition of reset event now allows them to be specified as such.

3.1.1 Bracket Notations

Existing compound ECS TEs have no well-defined reset event for various parts of the equation, and can be ambiguous when implemented, for example the TE

$$\partial out \leftarrow \partial in_1 \cdot \partial in_2 \cdot \partial in_3$$

could be implemented as 2×2-input *Last* gates, or a 1×3-input *last* gate. The temporal behaviour of these two choices *is* different, and could have subtle effects on how the system operates and the detection of errors. Therefore, two types of brackets are defined, non-persistent and persistent, that allow the definition of reset events to be made in all TEs and also allow disambiguation of TEs like those just described.

The Non-Persistent Bracket – ()

The non-persistent bracket () does not *persist* after its contained events have enabled, therefore the bracket is defined such that the output event for the bracket is consistent with the definition. When the expression contained in the () bracket pair enables, it is *held* true until the event considered the reset event outside the bracket enables. For example, in the TE

$$\partial \hat{out} \leftarrow (\partial in_a + \partial in_b) \cdot Ready$$

when ∂in_a or ∂in_b occurs and *Ready* is not true, the relevant enabling input event *stays true* until the reset event outside the bracket, $\partial \hat{out}$, enables. This is the most restrictive type of grouping that may be made, since it demands that essentially *all* inputs remain true until their final reset event occurs.

The Persistent Bracket –[]

The persistent bracket [] *persists* after its contained events have enabled, and thus functions as a *pseudo* output event for the contained expression. Using the previous TE in this form,

$$\partial \hat{out} \leftarrow [\partial in_a + \partial in_b] \cdot Ready$$

In this case, when either ∂in_a or ∂in_b become true, the bracket is enabled, and functions as the reset event for the contained variables, and the relevant input event resets. The gate output event, $\partial \hat{out}$, functions as a reset event for the reset of the TE, including the implied output event of the [] bracket pair. Although in this case the allowed behaviours using () and [] brackets are identical, if the two events were ANDed instead of ORed, then the allowable temporal behaviours of the two forms would be different. If the contained expression contains a reset event, then the [] pair does not function as a pseudo-reset event for the contained expression, but does still function as an output event.

3.1.2 The *Until* and *Latch* Gates

The *until* gate is a special aspect of the ECS representation, primarily because it interfaces between the two-phase *event-oriented* domain and the logical *level-oriented* domain. However, the definition of the temporal transform states that a temporal event variable remains true *until* its corresponding output event occurs. In most gates, the event signal which functions as the reset event for signals in the gate equation is identical for all signals, however in the until gate this is not the case. Therefore, the notation for *until*, U, is retained. One alternative would be to explicitly name the reset event for *every* temporal variable, however this would lead to an unnecessarily crowded and obtuse gate representation.

The elimination of the conditional operator, $>$, means that the latch gate must now be specified in an alternative way. In ECS′, the latch gate is specified

$$out \leftarrow in \cdot latch + out \cdot \overline{latch}$$

which is a full boolean expansion of the logical latch function.

3.1.3 Allowable TEs

Now that the reset event of a TE is explicit, a rule defining the form of *allowable* TEs can be given. Since the $\partial\hat{reset}$ event in a TE resets both the input and output *events* when the TE evaluates true, it must be the case that the reset event *must* be able to *reset* the truth of a TE, once it evaluates true. Therefore, TEs like

$$\partial\hat{out} \leftarrow \partial in_A + B$$

are not correct, because if B is true, the expression is true, but the output event $\partial\hat{out}$ has no mechanism for setting the truth of the temporal expression *false* since $\bigtriangledown B$ is not under control of $\partial\hat{out}$.

3.2 Event Flow Graphs

STGs and related graph-based techniques [Chu87] are powerful as a descriptive tool because circuit activity can be visualised, and circuit operation can be described and simulated using graph-based algorithms. Therefore, some similar method of graphical description for ECS circuits is desirable.

The notational change to almost entirely AND and OR-based dependencies in gate operators makes the representation highly amenable to graph-based descriptions. These descriptions in ECS′ are termed *Event Flow Graphs* (EFGs), which show the flow of event dependencies and event resets in networks of gates. EFGs are used in this work to give a graph-based description of critical control circuits, so that the flow of control can be readily visualised, and are consistent with the ECS′ representation.

3.2.1 EFG notation

An EFG consists of a set of M nodes

$$\mathcal{N} = \{n_1, n_2, \ldots, n_M\}$$

that form the events and logical transition variables of the system to be described. This set of nodes $\mathcal{N}$ also has a marking set $\mathcal{N}_A$, containing boolean values indicating the *status*, or marking, of the nodes in $\mathcal{N}$. There are two separate sets of edges, $\mathcal{D}$ and $\mathcal{R}$. The elements of $\mathcal{D}$ indicate *dependency edges*, where each element of $\mathcal{D}$ is an ordered pair containing the start and end node of the edge,

$$\mathcal{D} = \{d_i \,|\, d_i = (n_k, n_j) \text{ where } k \neq j \text{ and } n_k, n_j \in \mathcal{N}\}$$

n_i is the start (source) node of the edge, with n_j being the end (sink) node, and a node cannot depend upon itself. The set $\mathcal{R}$ is the set of *early edges* which control the resetting of node markings. The term *early* edge is carried over from past work on $\mathcal{TTG}$s, STG-like descriptions of ECS networks [MA93]. An early edge is specified by an ordered triple — the start and end nodes, and a *marking* of the edge e_{m_i} (note that the edges in $\mathcal{D}$ do not have a marking value), thus

$$\mathcal{R} = \{e_i \,|\, e_i = (n_k, n_j, e_{m_i}) \text{ where } n_k, n_j \in \mathcal{D}\}$$

There is no restriction on the values of k and j for each early edge, as they may be identical in some circumstances.

3.2.2 EFG operation

The operation of the EFG is defined through two *firing* rules, formalising the actions taken at for a given graph status. The marking of a node n_i is denoted $\omega(n_i)$, and the marking of an early edge e_i is denoted $\omega(e_i)$.

Definition 3.1 (Node Activation)
The dependence set *of nodes, $\mathcal{N}_{D_i}$, for a node n_i is the set of all nodes that source an edge in $\mathcal{D}$ that terminates on n_i,*

$$\mathcal{N}_{D_i} = \{n_k | d_x = (n_k, n_i) \in \mathcal{D}\}$$

then

$$\omega(n_i) = \prod_{\forall n_k \in \mathcal{N}_{D_i}} \omega(n_k)$$

Definition 3.2 (Early Edge Firing)
If $\mathcal{R}_{S_i}$ is set of early arcs terminating *on node n_i, and $\mathcal{R}_{O_i}$ is set of early arcs* emanating *from node n_i,*

$$\begin{aligned} \mathcal{R}_{S_i} &= \{e_k | e_k = (n_x, n_i, e_{m_k}) \in \mathcal{R} \ \ where\ n_x \in \mathcal{N}\} \\ \mathcal{R}_{O_i} &= \{e_j | e_j = (n_i, n_x, e_{m_j}) \in \mathcal{R} \ \ where\ n_x \in \mathcal{N}\} \end{aligned}$$

then early edge processing is enabled at node n_i when

$$\omega(n_i) \prod_{\forall e_j \in \mathcal{R}_{S_i}} \omega(e_j) = 1$$

The marking of the enabled node is removed, and all output early arcs of the enabled node are marked,

$$\begin{aligned} \omega(e_j) &= 0 \quad \forall \quad e_j \in \mathcal{R}_{S_i} \\ \omega(e_k) &= 1 \quad \forall \quad e_k \in \mathcal{R}_{O_i} \\ \omega(n_i) &= 0 \end{aligned}$$

Definition 3.3 (Initial EFG state)

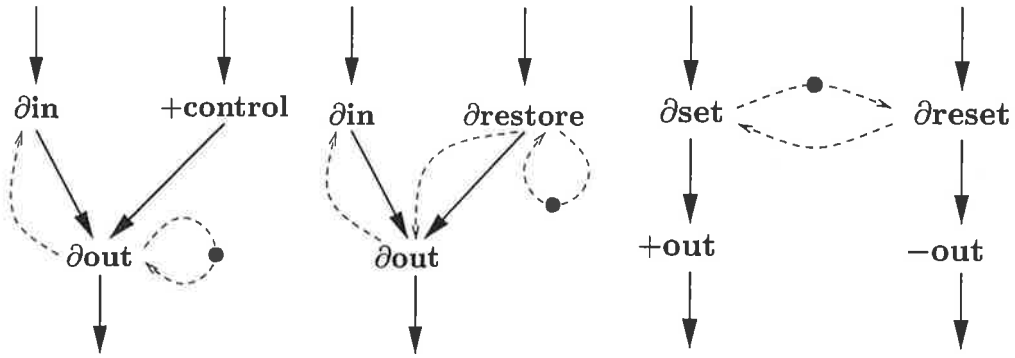
1. *Any self-terminating early edge is initially marked.*
2. *Any initialised event has its corresponding node in $\mathcal{N}$ marked.*
3. *Any until gate input initialised has its input early arc initially marked.*

When processing these rules in the evaluation of the state of an EFG, the *Node Activation* rule is processed first, withholding any updates to $\mathcal{N}_m$ until the *Early Edge Firing* rule has been evaluated for all enabled nodes.

The operation of boolean signals is similar to Definition 3.1. When all input nodes to the current node are marked, then this node itself is marked. EFGs will always contain complementary node definitions (both *boolean+* and *boolean−*, or multiple copies of both), and when one boolean *state* enables, it is expected that

1. Every copy of the enabling boolean signals' *state* is marked, and
2. every copy of the complementary state boolean signal is un-marked.

This ensures that booleans operate as would normally be expected.



(a) Send EFG (b) Restore EFG (c) Until EFG

Figure 3.7 EFG gate examples.

3.2.3 EFG Examples

EFGs of three ECS gates are shown in Figure 3.7. Dependency edges are drawn as solid lines, while *early* edges are drawn as dotted lines.

The EFG for the *send* gate, Figure 3.7(a), operates as follows. When nodes ∂in and $+control$ are marked, the node activation rule for node ∂out is enabled, and node ∂out is marked. When node ∂out becomes marked, the *early firing* rule enables, and early tokens are placed on all early edges, and the node ∂out is un-marked. Early processing is then enabled on the node ∂in (since there is only one early edge input), and node ∂in is un-marked. Note that there is no early edge from ∂out to $+control$, since it is a boolean signal.

The *restore* gate EFG is different because the output and reset events of the gate are not identical. In this case, when $\partial \hat{restore}$ occurs, if the node ∂in is not marked, then node activation on node ∂out does not occur, and early processing from node $\partial \hat{restore}$ is initiated. When ∂in does occur, the next time $\partial \hat{restore}$ occurs, the node ∂out is enabled and marked, and the $\partial \hat{restore}$ event also provides for gate reset in this case.

These two examples have not considered the connectivity *beyond* the level of just the singular gate equation. When this is considered, the only change ever made is the insertion of one extra *early* edge to the output event, if required. This shall be seen in more detail in Section 3.2.4.

The Until gate EFG of Figure 3.7(c) is not totally consistent with the representation $out \leftarrow \partial set\, U\, \underline{\partial reset}$. The EFG instantly *resets* the ∂set and $\partial reset$ nodes whenever their respective transition occurs, which is inconsistent with the full temporal behaviour. A full expansion of the until EFG is shown in Figure 3.8.

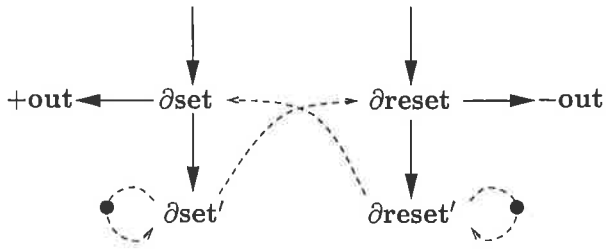


Figure 3.8 Exhaustive *Until* EFG.

This is obviously more complicated and unwieldy than the EFG of Figure 3.7(c), however, it does implement a behaviour consistent with the representation based on the firing rules described. If the *alternative* form of behaviour for the *until*, shown in Figure 3.7(c), is acceptable, then it can be used as a more intuitive representation.

3.2.4 Conversion

The conversion of a general set of temporal equations (TEs) in ECS′ to an EFG can now be considered. For any reset event node, $\partial\hat{reset}$, a self-terminating initialised early edge is drawn to the node, and also (if necessary) to the output node of the equation. Any output event node in a TE also becomes the source for early edges to all input nodes that do not have self-terminating early edges. OR dependencies (e.g. $\partial out \leftarrow \partial in\, a + \partial in\, b$) require that a *copy* of the output node be created for each signal in the OR dependency apart from the first input, and each input to the OR feeds one such copy, with appropriate early edges inserted. *Until* components of TEs are simply replaced with an EFG like that shown in Figure 3.7(c).

Bracket notations in a TE require consideration. The use of () brackets implies that the internally-enabled expression should not *reset* until the reset event outside the brackets is enabled. Thus, when a () bracket pair is encountered in a TE, a *dummy* event is created for the output of the bracket, and this dummy event has early arcs returning from every event node that it feeds. The bracket pair [] must be treated differently because it is *persistent*. In this case, two dummy nodes must be created, with one for the contained expression's output, and the other to *retain* the output until the expression outside the bracket resets. This expansion process is shown in Figure 3.9.

An example is the conversion of the TE

$$\partial\hat{out} \leftarrow (\partial in_A \cdot (\partial in_D + \partial in_B)) \cdot Ready \cdot [\partial in_C \cdot \partial\hat{restore}_C]$$

to an EFG form, shown in Figure 3.10. Dummy events are created for the two bracket pairs () and [], being ∂X_2 and ∂X_3, respectively. These two dummy events feed the output node ∂out, and since ∂out is a reset event, a self-terminating initialised early edge is drawn to that

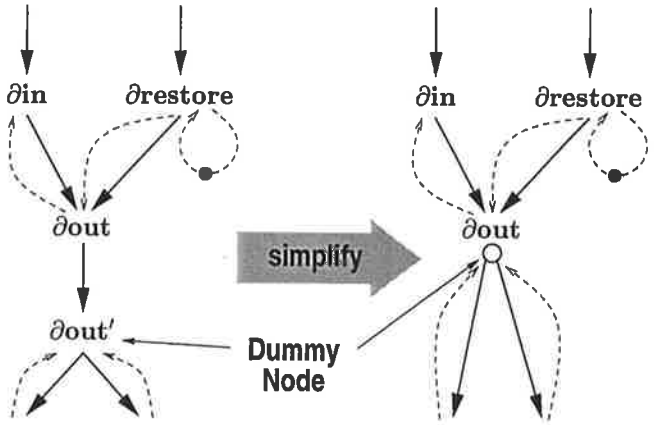


Figure 3.9 Output Event Distribution. When a persistent bracket pair, [ ], is used in a TE, in this case $[\partial in \cdot \partial \hat{restore}]$, the resulting EFG must ensure that the output of the bracket persists using a *dummy node*, for which a shortened notation can be used to improve clarity.

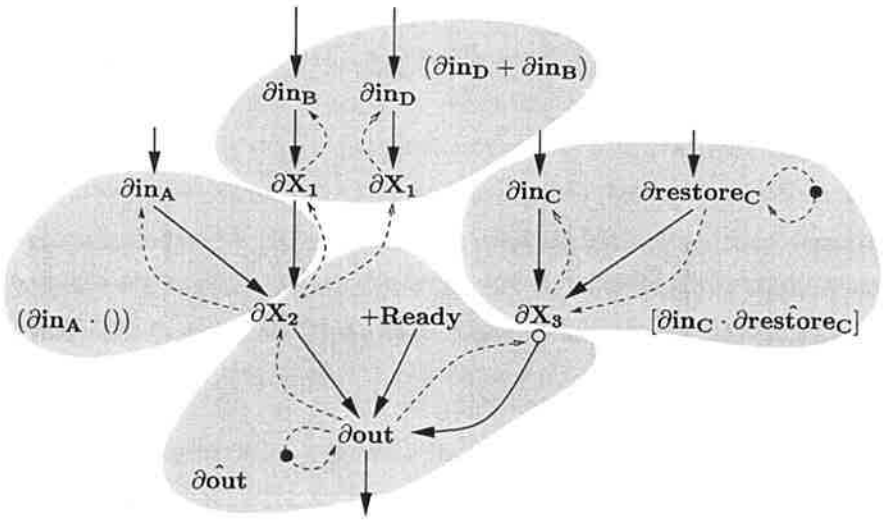


Figure 3.10 EFG Conversion Example.

node. The () bracket pair is now examined. This contains an AND dependence and another bracket, so another dummy event, ∂X_1, is created, and the AND dependence realised, with appropriate early edges added. The next () bracket pair is then examined, containing an OR dependence, requiring a copy of the output node to be instantiated, ∂X_1.

The [] bracket pair requires an output node like that shown in Figure 3.9. Examining the internals of the bracket pair shows an AND dependence. The event $\partial \hat{restore}_C$ is a reset event, so a self-terminating early edge is drawn to that node. An early edge from the dummy node ∂X_3 to the node ∂in_C is drawn to complete the conversion of the *restore* TE, $[\partial in_C \cdot \partial \hat{restore}_C]$.

4 Conclusion

Event Controlled Systems (ECS) is a bounded-delay asynchronous design methodology that generalises the two-phase design paradigm with a specification mechanism and a simple formalism that allows intuitive descriptions of gates and networks of gates, and the simulation of networks of gates and error detection during such simulations. Syntactical ambiguities are eased with the introduction of ECS′, a modification to ECS that improves the consistency of the representation, and enables a graph-based system description tool, EFGs, to be used for complex ECS gate networks.

ECS is designed for simple and intuitive representation of two-phase bounded-delay asynchronous systems, as well as simulation of these systems and the detection of violations of the two-phase signalling protocol. ECS provides a number of atomic gates for the design of fast and efficient control networks, which are unchanged in the ECS′ representation, and is explicitly aimed at *engineered* approaches to VLSI systems design.

ECS is a stable approach that has been used in the design of a number of VLSI systems (see Chapters 4 and 5). ECS′, as a more consistent representation of two-phase systems, will now be used as a basis for tool development and more work on formal techniques and algorithms for this approach in the future.

This chapter has only discussed aspects of the *representation* of two-phase asynchronous systems. The rest of this work will focus on the *design* and *performance* of systems using the ECS approach.

Notes

1. The somewhat unfortunate term *temporal transform* was not changed (although this could infer that this is a class of *temporal logic*) to ensure consistency with previous ECS publications.
2. This differs from the "*voltage*" name used by Morton [Mor97]. The actual physical value of a signal is viewed as a zero or one before transformation to the *temporal* domain, considered a *logical* value.
3. This notation differs from that of Morton [Mor97], which uses $\overline{\partial signal}$ – this can easily be confused with boolean inversion, whereas using $\underline{\partial signal}$ has no such connotations.

Chapter 4

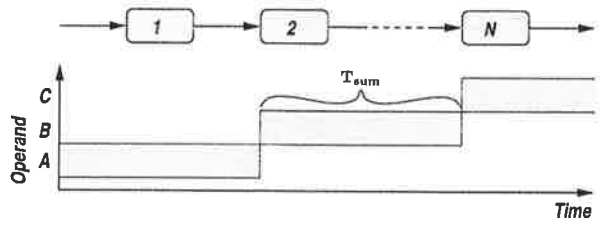
Pipelines

PIPELINING is a widely used technique [Kog81] that breaks sequentially-dependent processing into individual computations, and then computes step-by-step results in parallel, resulting in higher throughput. An N-step sequential task and its pipelined implementation are shown in Figure 4.1, where the pipelined implementation obtains higher throughput and logic utilisation at the expense of higher latency and area (due to the overhead introduced by the latches).

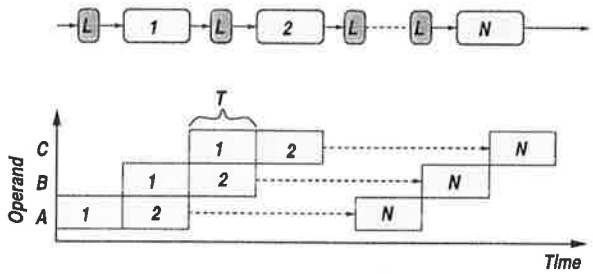
The choice of inter-stage synchronisation is not explicit in a pipelined system. Global synchronisation is conceptually very simple, has very little time overhead, and makes communication between non-adjacent stages relatively easy. However, maintaining exact synchronisation in a large, high-speed pipeline is a difficult design issue [Fri95, Ell97]. Asynchronous communication between stages involves a communication overhead and complicates communication between non-adjacent stages, but eliminates global timing problems. Such asynchronous pipelines are well explored [Sut89, Fur95, AML97c] as mechanisms for inter-stage synchronisation. The design, timing analysis, performance, and test of ECS pipeline structures is explored in this chapter. The *overriding* goal of the design of these ECS pipelines is to improve speed performance by using the bounded-delay model. This requires timing constraints to be defined such that these pipelines operate correctly, and the design issues in meeting these constraints are examined.

1 Pipeline Control

Asynchronous pipelines are conceptually elegant, since each stage communicates by *request*



(a) N-step process and Time Behaviour



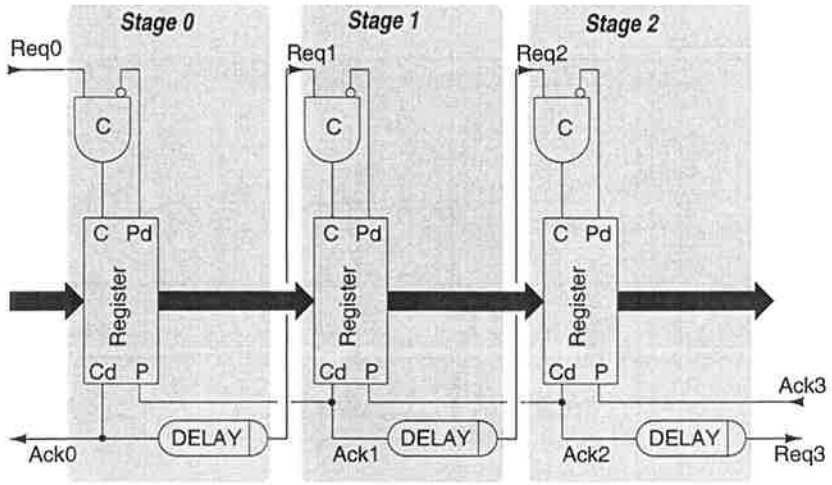
(b) Pipelined Implementation and Time Behaviour

Figure 4.1 Pipelining a N-step task. T_{sum} is the sum of the time taken to complete the N tasks, and is the maximum speed of the sequential process, (a). T_{worst} is the time taken to complete the slowest of the N tasks, and combined with latch delay, sets the maximum cycle time of the pipelined process, (b). The pipelined throughput is greater than the sequential process allows.

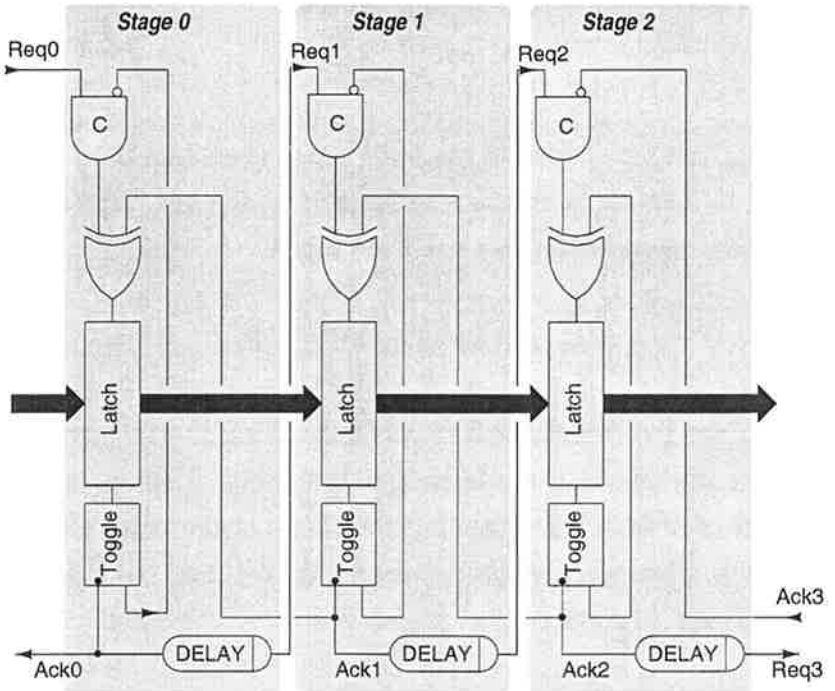
and *acknowledge* signals, regulating the state of the pipeline and eliminating the need for global synchronisation. Unfortunately, the localised control used to regulate the state of the pipeline is part of the critical path of each stage, causing a throughput degradation. This performance degradation should be minimised, in the case of ECS neccesitating a *minimum* of control on the critical path.

A well-known two-phase asynchronous pipeline structure is the *micropipeline*, developed by Sutherland [Sut89]. The purest form of such a pipeline is shown in Figure 4.2(a). The registers shown in Figure 4.2(a) are *capture-pass* registers. A transition on the *capture*(C) line causes the registers to close, providing the signal *capture-done*(Cd) when complete, and a transition on the *pass*(P) line causes the registers to open, providing *pass-done*(Pd) when complete. Although it is an elegant structure, the *capture-pass* registers tend to be large and slow, and the circuit of Figure 4.2(b) is used to enable the use of single-phase latches [DW95]. Note that the control circuit is speed-independent (excluding the delay to model the datapath latency).

The speed of the traditional *micropipeline* [Sut89] could be improved by simply removing one



(a) Micropipeline using *Capture-Pass* registers



(b) Micropipeline using single-phase latches

Figure 4.2 Micropipelines. The structure can employ special *event-controlled* registers, (a), or standard *single-phase* latches, (b).

of the artifacts of the speed-independent model, the *toggle* gate. Such a structure is shown in Figure 4.3, now using ECS gate notation.

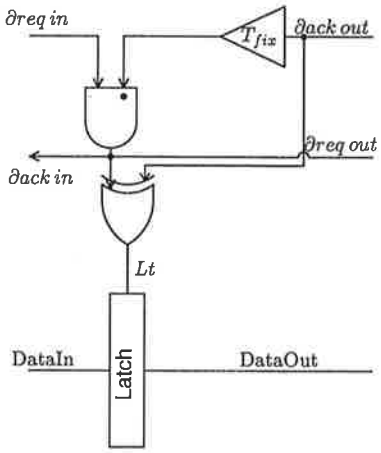


Figure 4.3 Bounded-Delay Micropipeline Circuit. Note the absence of the TOGGLE gate, even though single-phase latches are being used. The TOGGLE is replaced by timing constraints for the control path, and a delay T_{fix} to ensure an adequate latch pulse.

A delay element, T_{fix}, must be added in the path which re-enables the *Last* gate. The value of this delay should be such as to ensure a sufficiently wide pulse on the latch control line, Lt, before any pending *request* event can set it low again,

$$T_{fix} \geq T_{\uparrow Lt} - T_{\downarrow Lt} + T_{pl} - T_{last}$$

where $T_{\uparrow Lt}$ and $T_{\downarrow Lt}$ are the rise and fall times of the latch control signal Lt, respectively, and T_{pl} and T_{last} are the delays of the latch and last gates. This value of T_{fix} provides a latch-line pulse width of at least T_{pl}. Note that the design of this delay element changes with the width of the latch. If more gates are connected to Lt, then the designed value of T_{fix} may need to be modified.

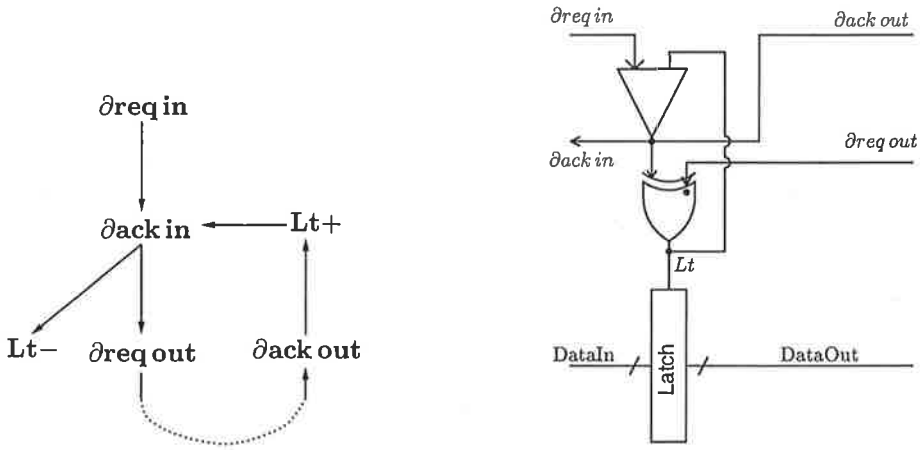
1.1 State Pipeline

The latch control line, Lt, functions as a pseudo-*stage busy* signal. When Lt is low, the latches are closed and the stage is busy, and when Lt is high the latches are open and the stage is free. Whenever the stage is *busy*, a new request should be stopped at the input. When the stage is *free*, a request can propagate though and cause the stage to latch and issue an acknowledge. Such a control schema can be implemented directly, and is known as the *State Pipeline* [MAL95, AML97b, AML97c] (hereafter referred to as the S-Pipe), shown

in Figure 4.4 as a FIFO (no processing delay inserted). The temporal equations governing its operation are

$$\begin{aligned} \partial \hat{ack}\, in &\leftarrow \partial req\, in \cdot Lt \\ Lt &\leftarrow \underline{\partial ack\, out}\ U\ \partial ack\, in \\ \partial \hat{req}\, out &\overset{T_d}{\leftarrow} \partial ack\, in \end{aligned}$$

where T_d is any delay required to match the latency of the datapath, and is not required when using this pipeline as a FIFO (and thus $T_d = 0$).



(a) Event Flow Graph

(b) Circuit Implementation

Figure 4.4 State Pipeline (S-Pipe). This circuit has no added delays for processing and is thus a FIFO. In (a), a simplified Event Flow Graph without early edges is shown, while (b) gives the circuit implementation.

The state pipeline does not need delay elements in the return $\partial ack\, out$ path as both the datapath latches and the *send* gate at the input are controlled by the same signal, Lt, and these gates are virtually identical (latch and *send* gate). However, the input acknowledge, $\partial ack\, in$, is issued *before* the latch control line has gone low, and this will obviously require timing validation.

Critical Path

The critical path control logic is shown in Figure 4.5. This path determines the maximum rate at which this pipeline structure can operate. The dependencies making up the critical path timing, T_{cp}, equate to

$$T_{cp} = 2 \cdot T_{send} + T_{until} + T_{\uparrow Lt} + T_{di}$$

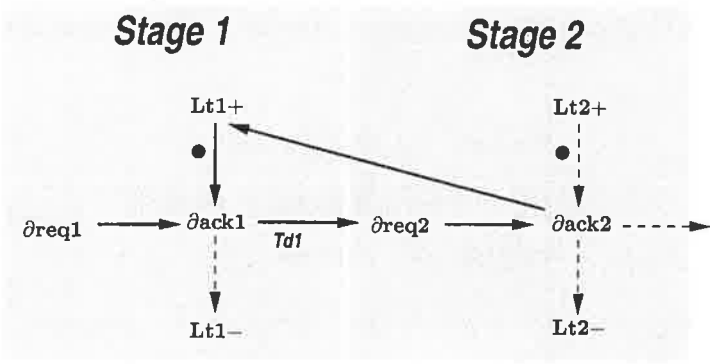


Figure 4.5 S-Pipe Critical Path. The critical path for the cycle time of the pipeline using a simplified EFG, not showing early edges. The critical path through the EFG is shown using solid edges.

where T_{di} is the inserted delay to model the datapath latency (and is thus the inserted delay between $\partial ack\,in$ and $\partial req\,out$). If the circuit is operating as a FIFO, then $T_{di} = 0$ and the minimum cycle time is

$$\begin{aligned} T_{cycle} \equiv T_{cp} &= 2 \cdot T_{send} + T_{until} + T_{\uparrow Lt} \\ &\approx 4.5\ \mathcal{T}_g \end{aligned}$$

where $\mathcal{T}_g$ is a gate delay in the target technology (see Appendix C).

1.1.1 Operating Constraints

The S-Pipe is a bounded-delay method of pipeline control, and the constraints which govern its operation are detailed below.

Response Time Constraint

The first constraint concerns the minimum interval between input request events. As the input acknowledge signal, $\partial ack\,in$, is issued before the latch control line, Lt, has gone low, it must be ensured that

$$\Delta T_{\partial ack\,in \rightarrow \partial req\,in} \geq T_{until} + T_{\downarrow Lt}$$

The time for the previous stage to receive the acknowledge and return a new request must be greater than the time required to set Lt low. Typically, $T_{until} + T_{\downarrow Lt}$ will be about $1.5 - 2.0$ gate delays, and any system generating data into a pipeline of this type would be hard-pressed to generate new data *and* send the request within this space of time.

Fast Path Constraint

The *Fast Path* constraint concerns the possibility of data *falling through* from one stage to the next, causing an error. An illustration of the problem is shown in Figure 4.6.

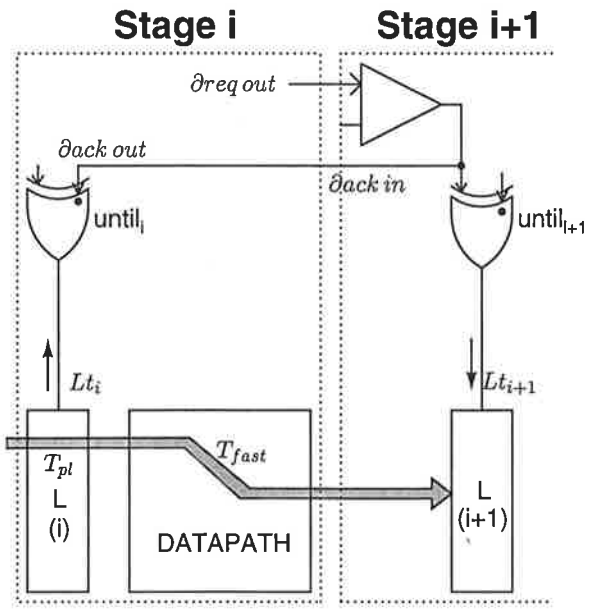


Figure 4.6 Fast Path Constraint. Data can potentially feed though prematurely from stage *i* to stage *i+1* as Lt_{i+1} is falling at the same time as Lt_i is rising.

The event *∂ack out* causes Lt_{i+1} to fall as well as Lt_i to rise, and it is possible that $\triangle Lt_i$ could occur well before $\triangledown Lt_{i+1}$, causing new data to fall through to the $i+1^{th}$ latch before it has closed, had timing issues not been addressed. Therefore, the circuit design must meet the constraint,

$$T_{until\ i+1} + T_{\downarrow Lt_{i+1}} \leq T_{until\ i} + T_{\uparrow Lt_i} + T_{pl} + T_{fast\,i}$$

This is assuming that both *∂ack out* and Lt_i are in equipotential regions (that is, they arrive at all parts of the circuit that they drive at the same time), with $T_{fast\,i}$ being the fastest path from the logic inputs to the outputs of the i^{th} stage. If we assume identical implementations for the *until* gates, then this becomes

$$T_{\downarrow Lt_{i+1}} \leq T_{\uparrow Lt_i} + T_{pl} + T_{fast\,i}$$

In the most restrictive case in which $T_{fast\,i} \approx 0$ (for example, a FIFO), then there is still over one gate delay margin (T_{pl}) between the delay of $\downarrow Lt_{i+1}$ and $\uparrow Lt_i$, making this a simple timing constraint to meet and check.

Event Timing Constraint

This constraint is a formalisation of the bundled-data constraint found in other methodologies. A generalised pipeline stage is shown in Figure 4.7. In any asynchronous pipeline, the forward control latency must be at least equal to the forward datapath latency (otherwise, control

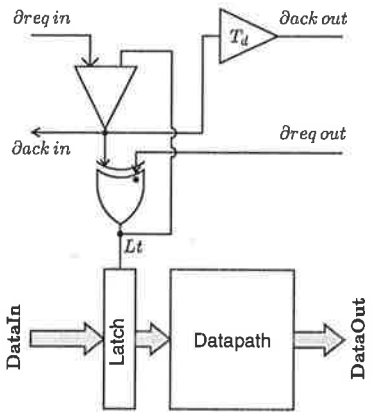


Figure 4.7 General S-Pipe Stage. In this case, the bundled-data constraint is satisfied by using a delay element to model the computation delay.

can get well ahead of data and cause incorrect operation at some later point in the system). In the case of the S-Pipe,

$$T_{send} + T_d \geq T_{pl} + T_{slow\,i}$$

where $T_{slow\,i}$ is the slowest propagation path through the datapath logic. However, the *send* gate is typically constructed identically to the latch (they are almost identical circuit-wise), and so the only constraint to be met is

$$T_d \geq T_{slow\,i}$$

The specification of control-data dependence uses the *Depends Upon* operator,

$$\partial \hat{req}\, out \leftarrow DataOut \;:\; \partial ack\, in$$

which specifies that $\partial req\, out$ does not occur after $\partial ack\, in$ until the data bus $DataOut$ is set properly, equivalent to the *bundled-data* constraint. This dependence is translated into the delay operator, shown as T_d in Figure 4.7.

1.2 Performance Comparisons

The performance of the S-Pipe can be compared to existing approaches [Sut89, DW95, FL96] for operation as a FIFO. Each pipeline controller circuit was extracted from 0.8μm CMOS [PP94] layouts and simulated using HSPICE [Met96]. The results are shown in Table 4.1.

The Micropipeline is shown in Figure 4.2(b), with the BD Micropipeline shown in Figure 4.3. The semi-decoupled four-phase (SD 4ϕ) controller [DW95] is fast, but not speed-independent, while the fully-decoupled (FD 4ϕ) controller [FL96] is speed-independent. The S-Pipe outperforms existing designs by margins of over 53% in the FIFO mode. It also improves on

Parameter	Micropipeline	BD Micropipeline	SD 4ϕ	FD 4ϕ	S-Pipe
Cycle Time	16.5ns	4.8ns	8.3ns	11.1ns	3.9ns
Latency	38.1ns	11.2ns	14.3ns	17.6ns	8.6ns
Restart	54.9ns	17.3ns	23.0ns	44.8ns	12.9ns
Power-Time Figures					
$P_{avg} \cdot T_{cycle}$	325	129	222	307	122
Figures of Merit					
n_{merit}	32.7	2.4	3.8	13.5	1.7

Table 4.1 FIFO Performance Comparisons. Results shown are for 5V operation at 75°C, using a Level 13 HSPICE model, with nominal parameter values [PP94]. Latency times are values for a 5-stage pipeline. Restart time is the time taken to restart the first stage after a *halting* condition is removed from the fifth stage. n_{merit}, a figure of merit, is $P_{avg} \cdot T_{cycle} \cdot A_{controller}$, where $A_{controller}$ is the area of the control component of each design, and P_{avg} is the average power consumption.

the cycle time of the bounded-delay micropipeline controller by 19%. Other four-phase and pulse-mode circuits [YBA96,MJCL97] have recently been devised, but their performance was not evaluated here.

1.3 Register Pipelines

The minimum cycle time of an arbitrary state pipeline stage is

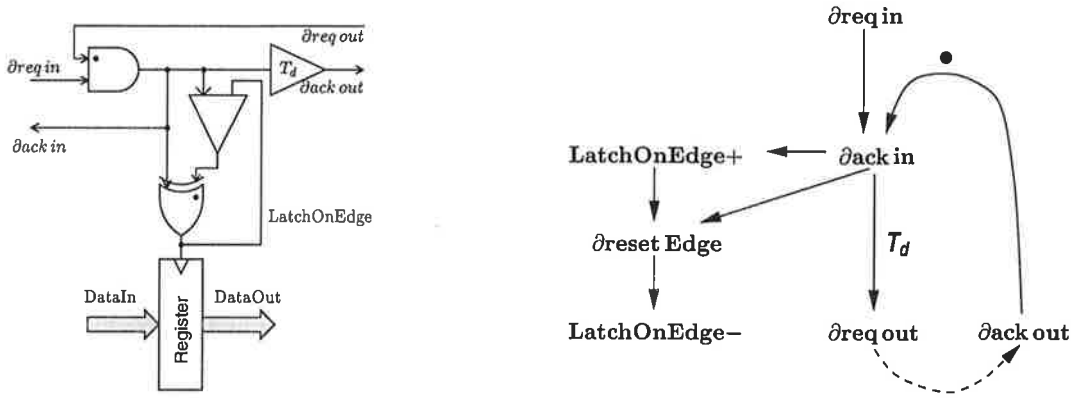
$$T_{cycle} \geq T_{pl} + T_{slow\,i} + T_{send} + T_{until} + T_{\uparrow Lt\,i}$$

and this is above the potential minimum of $T_{pl} + T_{slow\,i}$ that could be achieved by using a synchronous approach. This limitation comes from the need to keep Lt low during operation of the pipeline stage, and then set it high after $\partial ack\,out$ to latch data again. This problem can be overcome to some extent by using a pipeline with D-latch elements (registers) instead of level-sensitive latches, shown in Figure 4.8.

Inter-stage handshaking is now controlled by the *last* gate — using a *send* gate complicates the control for this class of structure. The input request, $\partial req\,in$, causes $\triangle LatchOnEdge$ when the *last* gate is primed (as it is initially). The *send* and *until* gates generate this required pulse. A returning acknowledge, $\partial ack\,out$, resets the *last* gate for a new request. The minimum cycle time of this configuration is

$$T_{cycle} \geq 2 \cdot T_{last} + T_{d\,i}$$

The control delays in the latch driver are *not* associated with this expression. The forward



(a) Circuit

(b) Simplified Event Flow Graph

Figure 4.8 D-Latch ECS Pipeline (D-Pipe). The driving delay of the latch control line, *LatchOnEdge*, is removed from the critical path and replaced with a constraint on the minimum value of the delay T_d. The *send* gate could be replaced with an appropriate *tap* out of the delay T_d to reduce the gate count. The EFG of (b) is simplified by not showing *early* edges.

control delay must still exceed the forward datapath latency,

$$T_{last} + T_d \geq T_{pl-reg} + T_{slow\,i} + T_{su}$$

where T_{pl-reg} is the D-latch propagation delay, and T_{su} is the required D-latch setup time. However, a certain minimum value of T_d is required to ensure that the latch *clocking* line, *LatchOnEdge*, has reset properly before the next $\partial ack\,in$ (which causes $\triangle LatchOnEdge$),

$$\begin{aligned} T_d &\geq 2 \cdot T_{until} + T_{\uparrow Lt_i} + T_{\downarrow Lt_i} + T_{send} - 2 \cdot T_{last} \\ &\approx 2.5\,\mathcal{T}_g \end{aligned}$$

This equates to a minimum *depth* of logic between the pipeline registers that should be used to avoid wasting the available computation time in each individual pipeline stage controlled in this manner. Using the above design parameters, the cycle time of the D-Pipe is

$$\begin{aligned} T_{cycle} &= 2 \cdot T_{last} + T_{d\,i} \\ &\geq T_{pl-reg} + T_{slow\,i} + T_{su} + T_{last} \\ &\qquad (datapath\ latency) \qquad (handshake\ overhead) \end{aligned}$$

For this design to improve upon the cycle time achievable using a S-Pipe,

$$\begin{aligned} &\text{D-Pipe} && \text{S-Pipe} \\ T_{pl-reg} + T_{su} + T_{last} &< T_{pl} + T_{send} + T_{until} + T_{\uparrow Lt} \end{aligned}$$

If $T_{pl-reg} \lesssim 2 \cdot T_{pl}$, the D-Pipe will improve on the S-Pipe. A *fast-path* constraint also exists, being

$$T_{last} + T_{pl-reg} + T_{fast\,i} \geq T_{hold-reg}$$

between two similar pipeline stages (such that time to drive register control line high is roughly equal), where $T_{hold-reg}$ is the hold time requirement on the register inputs after a "clock" transition. This constraint is certain to be satisfied due to the margin involved.

1.3.1 Pulsed-Latch Pipeline (P-Pipe)

It may be observed that we could use level-sensitive latches in an identical manner to the above, where the line *LatchOnEdge* performs an identical function to the line *Lt* in the S-Pipe. Such a structure is called the *Pulsed-Pipeline*(P-Pipe), and is identical to the D-Pipe, except that the edge-sensitive register is replaced with a level-sensitive latch (which uses control that makes it appear as a register).

The constraints for the pipeline are identical, and the delay element T_d has identical design parameters, except that the propagation and setup delays of the register (T_{pl-reg} and T_{su}) are replaced with the lumped propagation delay of the latch (T_{pl}), therefore to exceed the speed of the S-Pipe

$$\begin{array}{rcl} \text{P-Pipe} & & \text{S-Pipe} \\ T_{last} & < & T_{send} + T_{until} + T_{\uparrow Lt} \end{array}$$

which gives an improvement in cycle time of 0.9 gate delays when using a 32-bit latch. One further constraint is introduced due to the use of level-sensitive latches. As the latches *pass* data whenever *Lt* is high, a *fast-path* constraint exists (which was largely eliminated in the D-Pipe as the registers are edge-triggered),

$$\begin{array}{rcl} i^{th}\text{ stage} & & i-1^{th}\text{ stage} \\ 2 \cdot T_{until} + T_{send} + T_{\uparrow Lt_i} + T_{\downarrow Lt_i} & \leq & T_{last} + T_{until} + T_{\uparrow Lt_{i-1}} + T_{pl} + T_{fast\,i-1} \\ & \text{simplifying} & \Rightarrow \\ T_{fast\,i-1} & \geq & T_{until} + T_{\downarrow Lt\,i} - T_{last} \end{array}$$

This sets a requirement on the *minimum* logic depth between adjacent registers to avoid data feed-through without margin (recall that the S-Pipe had *margin* against feed-through, and the D-Pipe can usually ignore the problem as it uses edge-triggered registers).

1.3.2 Performance Comparisons

Performance comparisons with the S-Pipe are shown in Table 4.2. Note that both the P-Pipe and D-Pipe achieve approximately the same T_{prop} (available logic depth at this cycle

time) as the S-Pipe, but at an improved cycle time ($\approx$ 10% for this simulation). A dynamic D-latch [AY91] was used for the D-Pipe, which improved radically on the size and speed of existing designs [Puc90, WE95]. Despite the improved performance of the latch used, the D-Pipe consumes considerably more power due to the greater capacitive load on the register *clock* control line.

Parameter	P-Pipe	D-Pipe [AY91]	S-Pipe
Cycle Time	7.4ns	7.5ns	8.2ns
Latency	16.5ns	16.7ns	16.2ns
Restart	6.9ns	7.6ns	8.7ns
T_{prop}	4.4ns	4.4ns	4.3ns
Power-Time ($mW{\cdot}ns$)	110	458	101

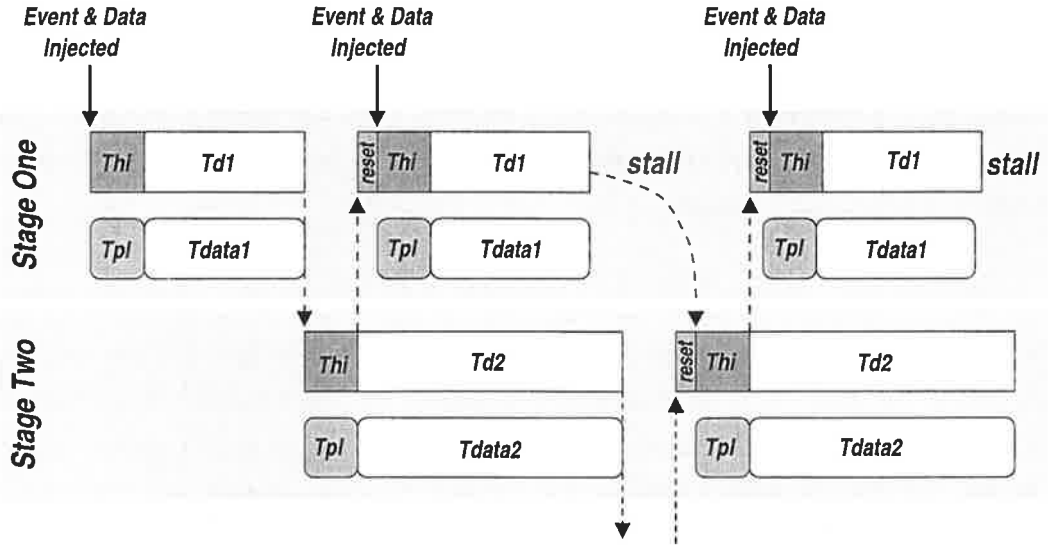
Table 4.2 D-Pipe and P-Pipe Performance Figures. All pipelines have an identical inserted delay element T_d=4.0ns (both the D-Pipe and P-Pipe are unable to operate as FIFOs). Results are given for a 3-stage pipeline simulated using a HSPICE Level 13 model with nominal parameters [PP94], at 75^{o}C with a 5V supply. Latency is propagation time from first stage *enabled* to the output request from the pipeline, while restart is the time taken to acknowledge a pending request from when a blocking condition is removed. T_{prop} is the available time in each stage for datapath computations. There is no latch output activity when comparing power consumption.

1.4 Control-Data Relationships in Asynchronous Pipelines

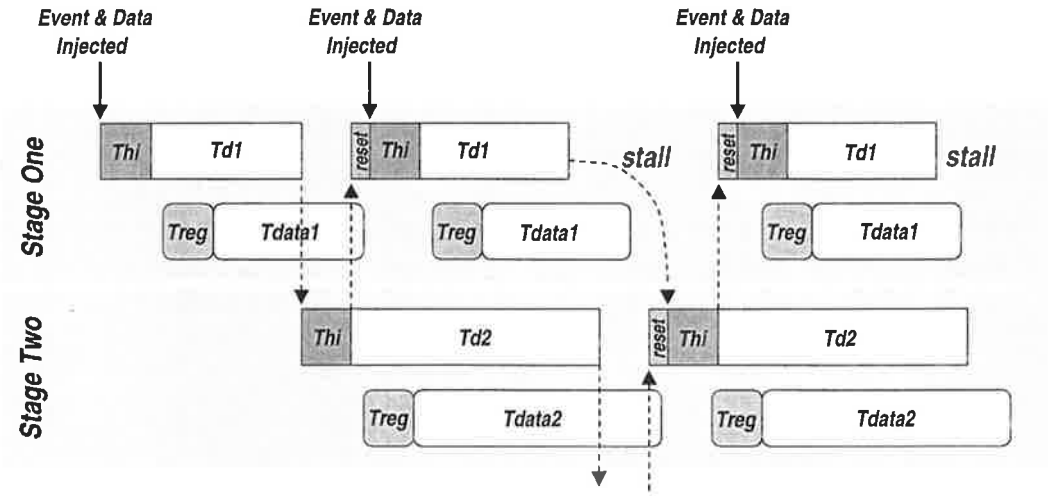
The timing relationships between control and data in the S-Pipe, D-Pipe, and P-Pipe vary considerably, due to the techniques used for latching. Even though in these structures the forward control latency is matched to the datapath latching and computational latency, this does not necessarily mean control *fronts* ($\partial req\ out$ in the S-Pipe) are synchronised with respect to data *fronts* ($DataOut$ in the S-Pipe). Timing relationships for the S-Pipe and D-Pipe are shown in Figure 4.9, where T_{hi} refers to the handshaking time spent between stages, and the $T_{data\,i}$ values are the forward control delays of each pipeline stage.

The S-Pipe tends to keep control in time-step with the data, since the forward paths are identical. Thus, the timing of Figure 4.9(a) shows that data outputs from a stage are ready at the time the output event is sent to the subsequent stage.

However, for the D-Pipe, data latching is not completed until well after the initial handshake action has completed, and thus the output data does not become valid until *after* the $\partial req\ out$ event is issued to the subsequent stage. This clearly violates the *bundled-data* constraint, but this is not technically an error since the assumption of self-similarity to the next pipeline



(a) Control-Data Timing in S-Pipe



(b) Control-Data Timing in D-Pipe

Figure 4.9 Control-Data Timing in ECS Pipelines. The *∂req out* events in S-Pipes, (a), tend to stay in synchronisation with the data fronts, but in D-Pipes, (b), data fronts will have to lag control to obtain optimal cycle time.

stage ensures that an error does not occur. Therefore, a way of expressing these relationships is required such that these situations can be handled and so that interfacing between different pipeline control types can be checked and handled without excessive manually-driven timing analysis.

1.4.1 Representing Dependencies

The assignment of gate delays to temporal equations (TEs) has already been demonstrated. However, the current method of expressing dependencies demands that the bundled-data constraint is satisfied, for example

$$\partial \hat{req}\, out \leftarrow DataOut \;:\; \partial ack\, in$$

for the output of a typical pipeline stage, whereas both the D-Pipe, P-Pipe, and to some extent the S-Pipe can break this assumption, within certain bounds. These bounds are defined by the tolerance at the input side of the pipeline stage to skew between control and data fronts, and at the output side by the relation between control and data delays. A mechanism is needed by which the relationships between signals can be made explicit, and explicit timing information added for this very purpose.

The Bind Operator

The *Bind* operator($\circ$) specifies a *dependency* or *association* between two signals, to which timing information can be added,

$$signal{-}1 \circ signal{-}2$$

The binding operator without any parameters merely makes explicit any dependency between $signal{-}1$ and $signal{-}2$, and has a precedence in between AND($\cdot$) and OR($+$) in a temporal equation. However, the general form of the *bind* operator is

$$signal{-}1 \overset{x|y}{\circ} signal{-}2$$

where x represents the value of the timing skew (skew being time difference) between $signal{-}1$ and $signal{-}2$ (where $signal{-}1$ leads $signal{-}2$ in time always), and y is the required upper bound on the value of the skew. The bind operator only ever relates to the left-most operator in the expression (and thus is not associative), and multiple binds in the same expression all relate to the left-most variable ($signal{-}1$). If no constraint on the value of the timing skew between the two signals exists, the binding is specified

$$signal{-}1 \overset{x}{\circ} signal{-}2$$

and

$$\mathcal{B}(signal{-}1 \overset{x}{\circ} signal{-}2) = x$$

to explicitly specify the timing skew between the associated signals ($\overset{|y}{\circ}$ indicates a constraint only without a timing value assigned as yet), with the function $\mathcal{B}()$ returning the value of the timing skew in the binding $signal{-}1 \circ signal{-}2$. The evaluation of the temporal value of bound variables proceeds by eliminating the rightmost signals in the expression until only one remains

$$\begin{aligned} &signal \circ signal_1 \circ signal_2 \circ \ldots \circ signal_n \Rightarrow \\ &signal \circ signal_1 \circ signal_2 \circ \ldots \circ signal_{n-1} \Rightarrow \\ &signal \end{aligned}$$

A negative value of the timing skew value (x) is *always* evaluated to zero (since a zero value corresponds to the equivalent of the bundled-data constraint, and relying on data arriving earlier than control can lead to errors in asynchronous pipelined systems). The bind operator is a much more flexible operator for the expressing the associations between signals because it allows for a wider range of timing behaviours in signals and interfaces, since the *Depends-Upon* operator is equivalent to $\overset{0}{\circ}$. As shall be seen, the general form of the *bind* operator can be used to detect *bind violations* in communicating modules.

1.4.2 Control Domain Interfacing

The bind operator can be used to make explicit timing relations in ECS modules, and here pipeline compatibility will be examined. The *compatibility* between pipeline stages (the ability for them to interface without timing errors) can then be examined based on relationships at pipeline inputs and outputs defined by the bind operator. The methods of control of asynchronous structures (including pipelines) will vary, and the bind operator is a way of both expressing and evaluating the timing of associations.

S-Pipe control-data timing

The S-Pipe control equations can have explicit associations added using the bind operator. One equation will constrain the input binding value, and the other gives the bind timing at the stage output,

$$\begin{aligned} \partial \hat{ack}\, in \leftarrow Lt \quad &\cdot \quad (\partial req\, in \overset{|c}{\circ} DataIn) \\ \partial \hat{req}\, out \overset{v}{\circ} DataOut \quad &\overset{T_d}{\leftarrow} \quad \partial ack\, in \end{aligned}$$

The first equation constraints the value of timing skew at the input to be within the value c, while the second gives the *value* of timing skew at the output of the stage. No constraint is added to the output binding, $\partial req\, out \overset{v}{\circ} DataOut$, because this constraint is dependent on the construction of the succeeding stage. The constraint c on the input binding and the

value v of the output binding are

$$\begin{aligned} c &= T_{send} + T_{until} - T_{pl} \\ v &= \mathcal{B}(\partial req\,in \circ DataIn) + T_{pl} + T_{slow\,i} - (T_{send} + T_{di}) \end{aligned}$$

Using the value of T_d from Section 1.1, the additive term of v is zero (as would be expected), resulting in an output skew equal to the input skew. Note that overdesigning T_d decreases the timing skew v, easing timing margins (although the minimum value of v is zero).

D-Pipe and P-Pipe control-data timing

The D-Pipe and the P-Pipe have identical input and output descriptions (although their internal structure is not identical),

$$\begin{aligned} \partial \hat{ack}\,in &\leftarrow \underline{\partial ack\,out} \cdot (\partial req\,in \overset{|c}{\circ} DataIn) \\ \partial \hat{req}\,out \overset{v}{\circ} DataOut &\overset{T_d}{\leftarrow} \partial ack\,in \end{aligned}$$

In the case of the D-Pipe (see Section 1.3 on page 65), the c and v values are

$$\begin{aligned} c &= T_{last} + T_{until} + T_{\uparrow LatchOnEdge} - T_{su} \\ v &= T_{until} + T_{\uparrow LatchOnEdge} + T_{pl-reg} + T_{slow\,i} - T_{di} \\ &\quad \text{using } T_{di} \text{ design value} \Rightarrow \\ &= T_{until} + T_{\uparrow LatchOnEdge} + T_{last} - T_{su} \end{aligned}$$

Thus, the design of the T_d element for the minimum cycle time allows for no timing margin at the input of the pipeline (since $c = v$). Note that because the D-Pipe latches on an edge, there is no relation between the value of the input timing skew and the output timing skew.

The P-Pipe (see Section 1.3.1) has more complex constraints due to the length of the Lt pulse,

$$\begin{aligned} c &= T_{last} + 2 \cdot T_{until} + T_{\uparrow Lt} + T_{send} - T_{pl} \\ v &= max(\mathcal{B}(\partial req\,in \circ DataIn), T_{last} + T_{until} + T_{\uparrow Lt}) + T_{last} + T_{di} - (T_{pl} + T_{slow\,i}) \\ &\quad \text{using } T_{di} \text{ design value} \Rightarrow \\ &= max(\mathcal{B}(\partial req\,in \circ DataIn), T_{last} + T_{until} + T_{\uparrow Lt}) \end{aligned}$$

If T_{di} is designed nominally (apart from a certain minimum value added by the input latch controller) there is zero added skew, while there exists some slack in timing skew, whereas for the D-Pipe there is no timing margin when using a nominally designed delay T_{di}.

Generalised control-data timing

The general expression for the binding parameters in a pipelined system can be expressed using four timing values. $T_{first\,latch}$ is the earliest time after $\partial req\,in$ that data can propagate through the latch, $T_{last\,latch}$ is the time from $\partial req\,in$ to the last possible time that $DataIn$ can arrive such that it will still be latched, $T_{fwd\,control}$ is the forward control latency, and $T_{fwd\,data}$ is the forward datapath latency (including latch delays). If the input and output bindings in a stage or module are defined as

$$\begin{aligned} \partial req\,in &\overset{|c}{\circ} DataIn \\ \partial req\,out &\overset{v}{\circ} DataOut \\ \text{then} &\ldots \\ c &= T_{last\,latch} \\ v = T_{fwd\,data} - T_{fwd\,control} &+ max(T_{first\,latch}, \mathcal{B}(\partial req\,in \circ DataIn)) \end{aligned}$$

which can be used to evaluate the input constraint and output timing skew value for a general asynchronous control structure (not necessarily a pipeline interface), and then evaluate the timing at the interfaces of this module.

Interfacing and Bind Violations

The verification that a module interface meets necessary timing requirements can now be checked directly by using binding operator specifications. A module interface, like those of the pipelines discussed previously, is shown in Figure 4.10.

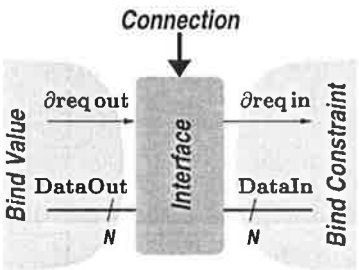


Figure 4.10 Interface Checking using the Bind Operator.

Verifying interface timing requires that the timing skew value at the $\partial req\,out$ side be assigned to the timing skew value at the $\partial req\,in$ side, and then performing a check for a constraint

violation on the $\partial req\,in$ side. Thus, examining the $\partial req\,in$ interface,

$$\begin{aligned} \text{if } \partial req\,in \overset{v_{in}|c}{\circ} & \; DataIn \text{ is the RH bind} \Rightarrow \\ \text{set } v_{in} = & \; \mathcal{B}(\partial req\,out \circ DataOut) \text{ and then} \\ \text{if } v_{in} > & \; c \Rightarrow \textit{Bind Violation!} \end{aligned}$$

Thus, if there is a connection in which the input side has a timing skew greater than the constraint at the output side allows, then this flags a bind violation. If no constraint violation is detected, then this allows the output timing skew of this module to be evaluated, now that knowledge of the input skew is available.

Cycle Stealing and and the $\mathcal{B}() \geq 0$ constraint

The bind operator will obviously allow only so much *under-design* of delays before bind violations start appearing due to control getting to far ahead of data. A possible technique in synchronous design is negative skewing or *cycle stealing* [Fri95], in which part of the clock period of one stage is stolen by another stage which has a slightly longer latency than the clock period. It may be envisaged that such a technique could be attempted in asynchronous pipelines also, resulting in one stage having a negative value of $\mathcal{B}(\partial req\,out \circ DataOut)$ and the immediately following stage *expecting* this and using it to complete its computation on data faster than the propagation delay of the local delay model would normally allow. However, if the pipeline halts or slows, then control ($\partial req\,out$) will catch data at the input to this stage, resulting in a zero-skew input and causing the stage to fail if it expected a large amount of negative skew (although small amounts could be tolerated according to the input constraint of the succeeding stage). Constraining the value $\mathcal{B}(\partial req\,out \circ DataOut)$ to be greater than zero (as was defined earlier) eliminates the potential for these failures, limiting the value of skew to be *at best* the bundled-data condition.

2 Pipeline Test

The autonomous nature of self-timed pipelines complicates the issues of test, because this autonomous nature makes it more difficult to assert global control for the purposes of internal signal observation. Two aspects of test can be envisaged - *functional* test attempts to ascertain that the logic is constructed and operating as required, while delay test determines whether the logic is operating at the desired speed. The test of the pipeline control logic, which is another facet entirely, is not examined here. Note that test functions are considered here as a separate mode from *at-speed* operation, and the control is designed to be initialised in *one mode only* and not to change without another initialisation cycle.

2.1 Functional Test

The test control logic for an ECS pipeline (based on the S-Pipe structure) is shown in Figure 4.11, which facilitates *scan test* [ABF90] at the outputs of the pipeline stage. Scan inputs could also come from the *DataLatched* bus if required.

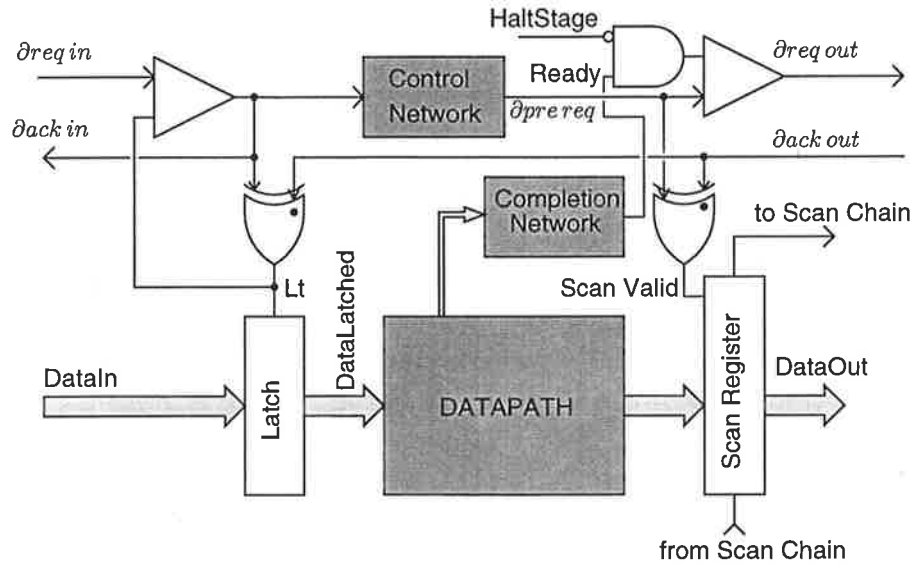
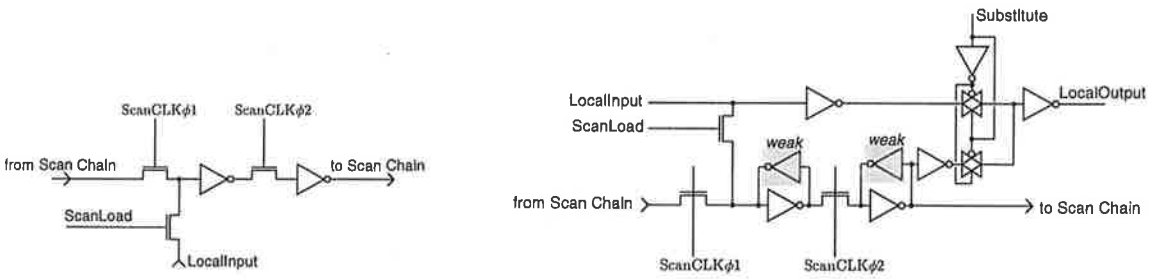


Figure 4.11 Pipeline Functional Test. The pipeline control is similar to a S-Pipe, with an output *send* gate to control when the output request event, *∂req out*, is issued.

The stage produces a completion signal, *Ready*. If the stage uses a delay model, then the AND gate and the completion network generating *Ready* is omitted. The scan test of the stage proceeds as follows. If the system is desired to be tested, then the *Halt* signal must be initialised high, otherwise the *Halt* signal stays low and the system runs *at-speed*. A request arriving at the stage will be latched and operated upon, but the *send* gate at the output stops the propagation of the output request event. Therefore, some safe time after the request event, the entire scan chain is loaded and the values shifted out for external observation. The signal *ScanValid*, also loaded into the scan chain, indicates that valid data from this stage was obtained (if no request is active in this stage at the time the scan chain is loaded, *ScanValid* will be low). Once complete, the *Halt* signal is pulsed to create a momentary high pulse on the signal which enables the output request, *∂req out* (thus the *Halt* signal is generated on-chip to enable suitable control of pulse width). The whole pipeline then advances one step — the pipeline is essentially being *clocked* at a low rate to facilitate scan (a similar but much simpler scheme was used for pipeline scan in *ECSTAC*, described in Chapter 5).

This scan logic has very low overhead when the pipeline operates *at-speed*, since the send gate which blocks the output request is normally present (as indicated by the *Completion Network* of Figure 4.11). Often the *Halt* signal can be pushed into the normal enabling of the *Ready* signal, adding no overhead for the addition of test control. In addition, the send gate which blocks the output can typically be squashed by using $Lt_{i+1} \cdot Ready \cdot \overline{Halt}$ as the enabling condition to the input send gate in the following stage (instead of just Lt_{i+1} as the enabling condition in the $i + 1^{th}$ stage).

Two possible cells used for the scan register are shown in Figure 4.12. A basic dynamic scan cell is shown in Figure 4.12(a), which is simply loaded and then clocked out using the two-phase non-overlapping clocks $ScanCLK\phi1$ and $ScanCLK\phi2$. In some applications, the outputs from a stage may be required to be set to facilitate fault coverage (and then latched by the subsequent stage in the scheme of Figure 4.11). The more complex scan register of Figure 4.12(b) can also set the register outputs by enabling *Substitute* and loading the scan chain with the data required for pipeline stage outputs (or inputs, depending on the placement of the scan register). The scan chain can then be loaded and scanned out, and this register incorporates a fully-static scan cell.



(a) Basic Scan Register Cell

(b) Fully-Static Loadable Scan Register

Figure 4.12 Scan Register Cells. ScanCLKϕ1 and ScanCLKϕ2 form a two-phase non-overlapping clock, where ScanCLKϕ1 should be in the low phase during scan register loading, when *ScanLoad* is high.

2.2 Delay Test

The testing of asynchronous pipelines for delay faults, especially pipelines heavily reliant on bounded-delay models (ECS pipelines in particular), is both extremely important and difficult. Synchronous pipelines can be delay tested by increasing the clock rate until the system fails, but this is not possible in asynchronous systems.

Testing an ECS pipeline stage involves ensuring that the stage sees incoming data and events identically to the *at-speed* case, letting the stage operate as per normal, and then latching the stage's outputs for observation. The control structure of some pipelines complicates the issue, as their input latches are *transparent* when the stage is not busy (for example, the S-Pipe).

Opaque Latch Scheme

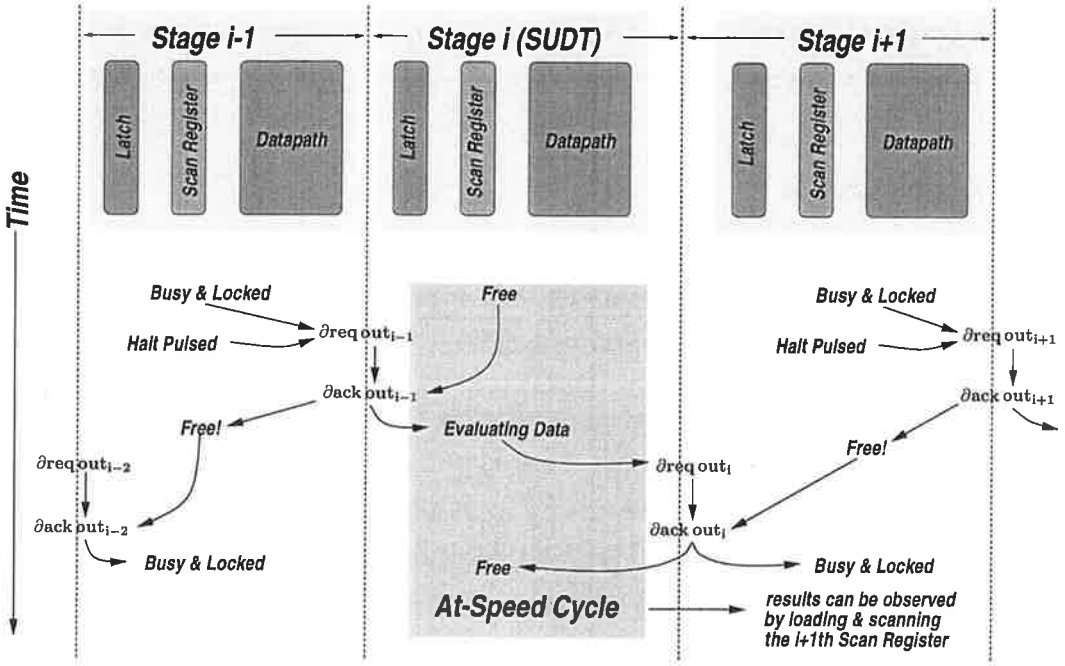
A method of delay testing a linear pipeline is shown in Figure 4.13, where any inactive stage is *opaque* (the latches/registers do not pass data when the stage is not busy). In this case, the datapath is assumed to be modelled by a delay.

As the latching elements are closed when the stage is free, an operand can be passed through the stage *at-speed*, and then the outputs of this stage latched and observed in the subsequent stage. The sequence of actions is shown in Figure 4.13(a). The design of the control logic integrates the requirements of functional and delay testing. When functional tests are required, both *TestOdd* and *TestEven* are raised, which halts every pipeline stage. When only *TestOdd* or *TestEven* are high, the pipeline is in delay test mode. Note that the placement of the scan registers is not identical to the functional test case (Figure 4.11). If the registers are moved to the outputs, then the delay test case requires more complex determination of expected outputs, and also does not permit direct observation of the output signals of the stage under test (which would permit determination of any delay-failing signals).

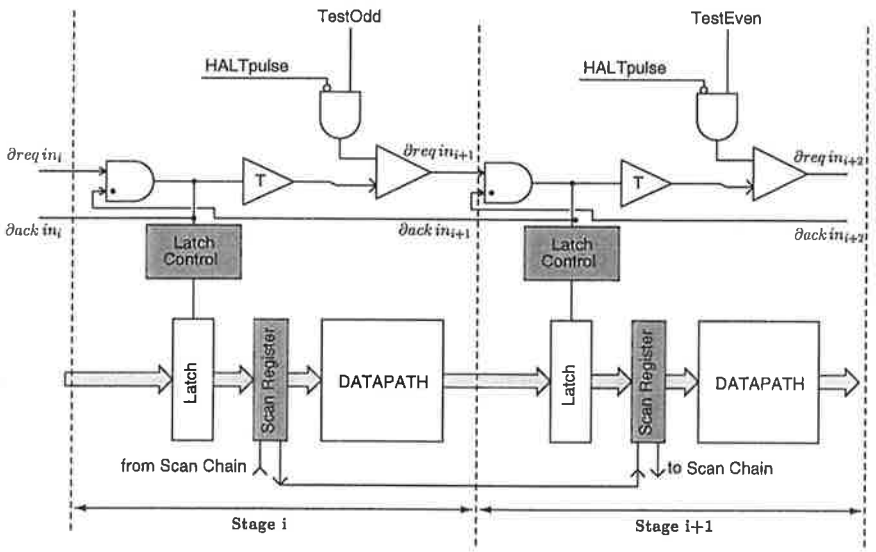
Transparent Latch Scheme

The *Opaque Latch* scheme cannot be used in pipelines which pass data in the inactive state (for example, the S-Pipe). Unfortunately, a modified version of the previous scheme which continues to use essentially global control of the scan chain cannot be used because possible variations in stage latencies (caused by data dependencies, for example) can prevent timing failures from being picked up in all possible cases. To provide a scheme which allows generalised pipeline test for delay faults in S-Pipelines, the control of the *TestLoad* signal is placed under local control.

In the i^{th} stage, the pipeline is blocked by raising $Halt_i$, and the pipeline banks up behind the i^{th} stage. The $Halt_i$ line is then pulsed to create a *bubble*, which propagates back up the pipeline. As the bubble propagates, each stage latches new data and computes new results based on this data at the *at-speed* rate before generating the $\partial req\,out$ event. This event is also sent to a control circuit, traversing an identical path to the normal latching control path used in each pipeline stage and shutting off the scan register inputs (and thus exactly mirrors the time taken to take the latch control line low in the subsequent stage when the stage is free). The signal *DelayTestActive* is set at initialisation if delay testing is required, and once

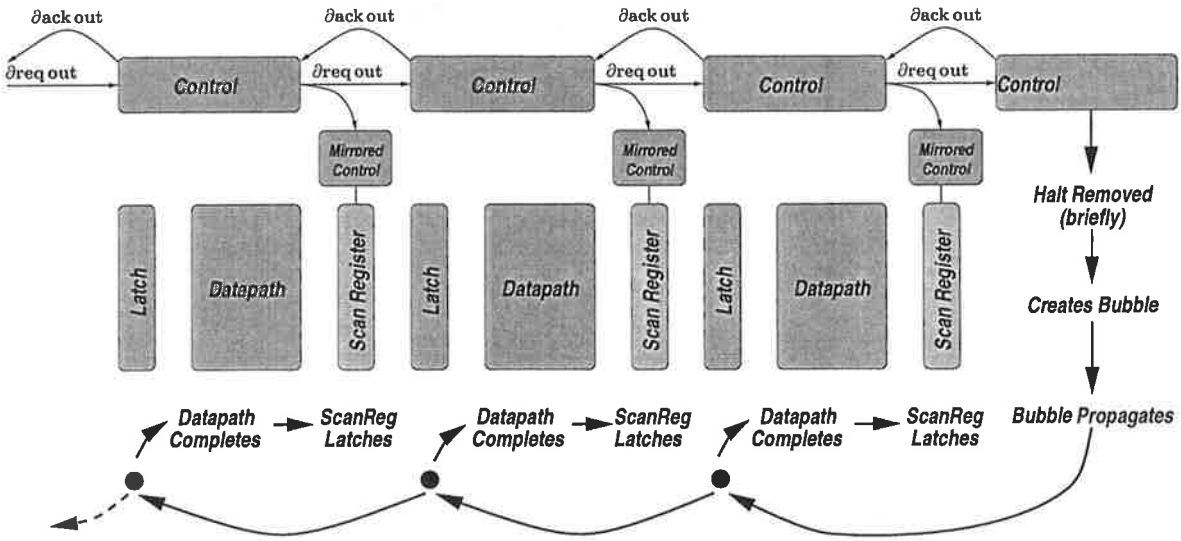


(a) Test Scheme

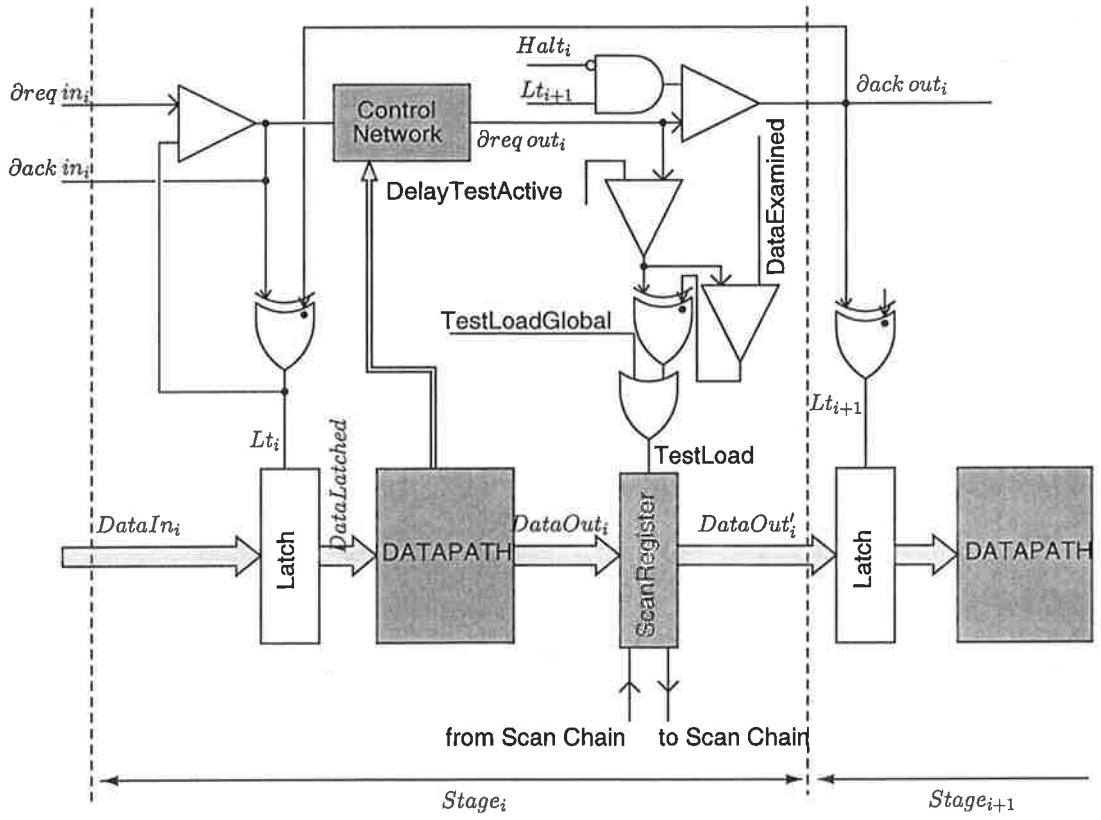


(b) Control Logic

Figure 4.13 Linear Pipeline Delay Testing. The scheme assumes that stages that are not busy *do not* pass data through the latching elements. The i^{th} stage is the stage-under-delay-test (SUDT), and every alternating stage is tested. The flow of control is shown in (a), with the necessary control logic shown in (b), using pipeline control like that used for the D-Pipe and P-Pipe.



(a) Delay Test Scheme for S-Pipes



(b) Control Signal Generation

Figure 4.14 Delay Testing the S-Pipe. The method is shown in (a), with the control circuit shown in (b). Note that one send gate has been eliminated from the control path (compare Figure 4.11).

the data stored in the scan chain from the last bubble have been examined, *DataExamined* is pulsed to open the scan registers again.

The pipeline can thus be halted for delay tests at the *end* of the pipeline, with the bubble propagating back up the pipeline, causing latching in the scan registers locally. Suitable time is alloted after the generation of a *Halt* pulse for all the stages to have operated before internal scan is commenced. Unfortunately, this brings back the control-data timing at all interfaces to be equivalent to the *bundled-data* constraint, which may cause this mechanism to fail to sample some timing faults with paths relying on specific input control-data bind timing.

This scheme can also be used to test *opaque* pipelines, but requires the localised *TestLoad* control to be modified to properly mirror the action undertaken by the subsequent stage's control. However, the *Opaque Latch* test mechanism for these types of pipelines does not require any additional local control.

3 Conclusion

This chapter has presented the design, analysis and test of asynchronous pipelines using the Event Controlled Systems approach. The principal ECS pipeline controller, the *State Pipeline* or S-Pipe, improves upon existing methods of asynchronous pipeline control by margins exceeding 50% when operating as a FIFO [AML97c]. The S-Pipe is a bounded-delay circuit, and the operational constraints and timing verification requirements were explored to show that they are relatively simple to meet, as they are all *single-sided*. Two alternative pipeline controllers, the D-Pipe and P-Pipe, improved slightly on the cycle time of the S-Pipe at the expense of power consumption and reduced timing margins, respectively. Analysis of the timing constraints of all three pipeline controllers has been presented.

Module interface timing requirements are clearly critical in a methodology that promotes the use of modular systems. Thus, a notational mechanism that makes control-data associations explicit was developed, and its use explored through the analysis of the interface timing requirements of the three pipeline structures described in this chapter.

Testing of asynchronous systems is an important issue. The functional and delay test of ECS pipelines was explored, and to maintain the potential performance gains realisable by using ECS, minimising the test circuits' impact on cycle time was a central concern. The circuits described do not, however, address the issue of test of the *control* components, and are purely for adding test abilities to the datapath of the pipeline. A simpler version of the functional test scheme described was used in ECSTAC, to be described in Chapter 5.

Chapter 5

The ECSTAC Microprocessor

ECSTAC is a pipelined asynchronous microprocessor designed to serve as a vehicle for the development of the *Event Controlled Systems* methodology of Chapter 3.

ECSTAC is a simple, structurally linear pipelined machine with an architecture similar to some early RISC machines [Hen84, Cho89]. The instruction set is custom defined (to suit the goals of our implementation), and the architecture was defined to fit in with cost constraints. The device implementation is moderately aggressive. Parts of the design use dynamic circuits to improve performance, but the majority of the design and all the control components use fully static logic. A microprocessor was chosen as a target for system implementation using ECS as it is not a control dominated architecture, but conversely the control is not trivial. Microprocessors also tend to aim for high performance, which integrated well with the stated target of ECS — to obtain higher performance from asynchronous logic.

The ECS approach had not been proven in a large-scale design, and the implementation of a complete microprocessor was expected to lend significant credibility to ECS as an asynchronous design approach. In addition, the issues of timing control and timing verification could be examined in the context of a large design, indicating how much effort the use of BD control would require. The design of ECSTAC also exposed a great many new ideas, techniques and methods for asynchronous design, one of the goals of the project.

1 Architectural Overview

The architecture definition stage revolved around the need to minimise the size of the die to be under some acceptable threshold. Therefore, it was decided that an 8-bit datapath core

would be used, with a 24-bit address path to permit a large range of test programs to be run. Instructions would consist of a variable number of bytes, depending on the particular requirements of the instruction.

No features were added that would unneccesarily complicate the architecture and would not be required in a prototype machine. ECSTAC does not have interrupts or exceptions, virtual memory support, floating point instructions, or integer multiply/divide. The core supports integer operations on 8-bit data only. The instruction set was designed to use 16 general-purpose 8-bit registers. However, because the address path is 24-bits wide, these registers can be grouped into four 32-bit register *quads* for address indexing. In addition, maintaining a stack pointer in one of these register quads would eliminate 25% of the register space instantly, so a separate stack pointer, and appropriate instructions to manipulate it, were defined as part of the architecture. Condition codes are set implicitly by any ALU operation into the *Flags register*(FR), which is used to control conditional branches.

The core was expected to operate at a reasonably high rate, faster than could conceivably be delivered from an off-chip memory system. Caches are used to serve the internal bandwidth requirements of the core. The pipelined nature of ECSTAC means that memory requests can come from two distinct points in the pipeline, the fetch stage and the memory stage. If a single, unified cache were used, this would add arbitration at every step to the cache critical path access time, which was deemed unacceptable from the start. Therefore, a split cache system was employed on ECSTAC, with an interface to external memory shared by both caches (which will require arbitration, but hopefully of a lower frequency).

1.1 Instruction Set

The instruction set architecture of ECSTAC consists of four groups of instructions — arithmetic/logical, program control, memory, and miscellaneous [MAJL94]. The ISA is summarised in Table 5.1.

1.2 Chip Architecture

A simplified block diagram of ECSTAC is shown in Figure 5.1. The pipeline is similar to a classic 5-stage RISC (*Fetch-Decode-Execute-Memory-Writeback*) [PH96] pipeline with a number of modifications that were required because of the architecture chosen.

The Program Counter (PC) and Instruction Cache (ICache) stages sequence the fetching of bytes into the Instruction Decoder (ID). The Instruction cache also interfaces with the Memory Unit (MU) when line fills are required.

The Instruction Decode unit performs the dual functions of instruction decode and latency

Form	Operation	Comments	S
Arithmetic/Logical instructions			
OP rd,rs_1,rs_2	$rd \leftarrow rs_1\ OP\ rs_2$		3
OP rd,rs_1,Imm_8	$rd \leftarrow rs_1\ OP\ Imm_8$	Imm_8 unsigned constant	3
OP rd,rd,rs	$rd \leftarrow rd\ OP\ rs$	Short ALU instn.'s	2
OP rd,rs,Imm_4	$rd \leftarrow rd\ OP\ Imm_4$	Imm_4 unsigned constant	2
Control Transfer Instructions			
CALL rs^Q	$Mem[SP] \leftarrow PC+1; PC \leftarrow rs^Q$	procedure call to reg.quad.	1
RETN	$PC \leftarrow Mem[SP]; SP \leftarrow SP-3$	return from procedure call	1
JUMP.cond rs^Q	$if(cond)\ PC \leftarrow rs^Q$	conditional jump to reg.quad.	2
JUMP.cond Imm_4	$if(cond)\ PC \leftarrow PC+$ SgnXtnd($Imm_4 << 4$)	conditional jump by *offset*	2
Memory Load/Store Instructions			
LD rd,rs_1^Q,rs_2^Q	$rd \leftarrow Mem[rs_1^Q + rs_2^Q]$	Load using two reg.quads.	2
LD rd,rs^Q,Imm_4	$rd \leftarrow Mem[rs^Q + Imm_4]$	Imm_4 unsigned constant	2
LD rd,Imm_{24}	$rd \leftarrow Mem[Imm_{24}]$	Imm_{24} data location	4
ST rd,rs_1^Q,rs_2^Q	$Mem[rs_1^Q + rs_2^Q] \leftarrow rd$	Store using two reg.quads.	2
ST rd,rs^Q,Imm_4	$Mem[rs^Q + Imm_4] \leftarrow rd$	Imm_4 unsigned constant	2
ST rd,Imm_{24}	$Mem[Imm_{24}] \leftarrow rd$	Imm_{24} data location	4
Stack and Miscellaneous Instructions			
PUSH rd	$SP \leftarrow SP+1; Mem[SP] \leftarrow rd$	Stack Push	1
POP rd	$rd \leftarrow Mem[SP]; SP \leftarrow SP-1$	Stack Pop	1
POPF	$FR \leftarrow Mem[SP]; SP \leftarrow SP-1$	Pop Flags	1
PUSHF	$SP \leftarrow SP+1; Mem[SP] \leftarrow FR$	Push Flags	1
FLSH	DCache flushed		1
IC DS/EN	ICache Off/On		1
DC DS/EN	DCache Off/On		1
NOOP			1

Table 5.1 ECSTAC Instruction Set. rs^Q indicates a register-quad group (four eight-bit registers) used for the purposes of generating a 24-bit address. The *S* field gives the size of the relevant instructions in bytes.

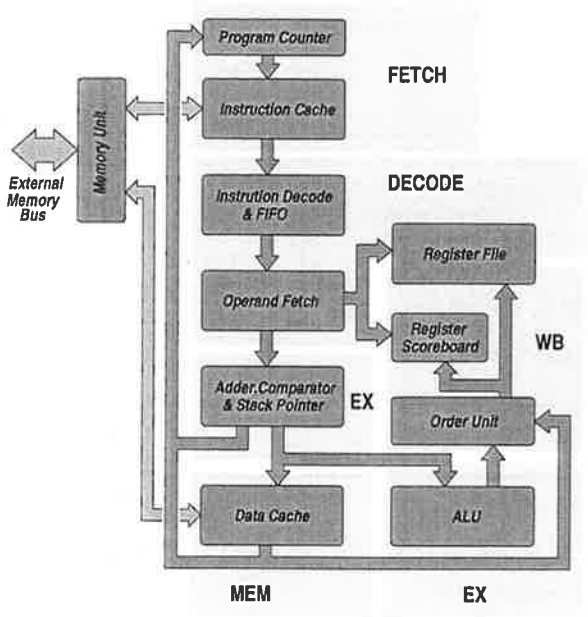


Figure 5.1 ECSTAC Block Architecture.

decoupling, integrating an 8-stage FIFO to absorb variations in the ICache access time. The Instruction Decode (ID) block feeds into the Operand Fetch (OF) stage, which fetches operands from the single read port register bank, and also stalls any instruction with an outstanding dependency. The register bank can fetch either a single byte-register, or a 24-bit register quad in one access. A register scoreboard is maintained which records the current destination registers outstanding in the pipeline after the OF stage, and when this register is attempted to be read, the read stalls until the dependency clears. When all dependencies for the current instruction clear, the OF unit issues the instruction to the ACS stage (Adder, Comparator, and Stack Pointer).

The ACS stage is an artifact of the chosen architecture. A stage which can perform the necessary 24-bit calculations for branches and stack pointer operations is required, since using the ALU with control added for three passes would have been very slow. The ACS handles the detection of taken branches and the subsequent updating of the program counter (as well as the control of the rest of the pipeline when a taken branch is detected). The unit also contains the stack pointer and all the control necessary for its use, including the logic necessary for CALL (which update the PC with a value, placing the old PC on the stack, requiring three data cache writes in sequence) and RETN (retrieving the old PC from three consecutive reads of the DCache, and updating the PC with this value) instructions.

The DCache serves all the memory requirements of the pipeline. It usually sends its output to the Order Unit (OU), but can also send its output to the Program Counter when this unit

is required to be reloaded from stack values.

The ALU is an eight-bit unit that handles the arithmetic, logical, and shifting instructions of the ISA. The output of the ALU is sent to the Order Unit. The Order Unit maintains in-order result writeback to the register file, ensuring that the earliest instruction to issue from the ACS stage writes its result back to the register file first.

Results arriving back from the OU write into the register bank, also resetting any scoreboard data and allowing dependent instructions to issue in the OF stage.

The processor was designed using a mix of full-custom and semi-custom/standard cell techniques. Custom layout, using the MAGIC tool [Adv95b], was employed for critical or heavily replicated datapath and array components to reduce area and improve speed. The design of the control and non-critical datapath elements was based around a custom-designed cell library for the $0.8\mu m$ CMOS process [PP94, MA94], with manual placement and routing of all parts of the chip.

2 Processor Design

The design of three of the main units of the processor pipeline will now be detailed. These units are the Instruction Cache (ICache), Data Cache (DCache), and Memory Unit (MU). The design of the execution core components [Mor97, Chapter 7] of ECSTAC was not the author's area of responsibility on this project, and is not detailed here.

2.1 Instruction Cache

A black box view of the ICache is shown in Figure 5.2.

The ICache receives a number of signals from the Program Counter(PC). $ICAddrIn$ is the 24-bit fetch address, while $PCcarry2$ and $PCreset$ are signals used to determine sequentiality of the address stream. $NoOpIn$ is the no-op signal used when the pipeline is being *annulled* due to a taken branch, with $NoOpOv$ resetting all of these signals in the pipeline. The ICache produces the address value used to initiate the fetch ($ICAddrOut$) and the corresponding data value, $ICDataOut$, at the output, as well as the latched $NoOp$ signal, for indicating a squashed instruction.

When the ICache requires a line fill from external memory, it places the address on $MU\ IC\ I/O$ bus and sends $\partial req\ MU$. The MU acknowledges at some later time, and subsequently sends a series of 32-bit values back to the ICache via $\partial MUdata$ and the $MU\ IC\ I/O$ bus. The number of words sent back is the value required to fill the ICache line (the MU knows the size of

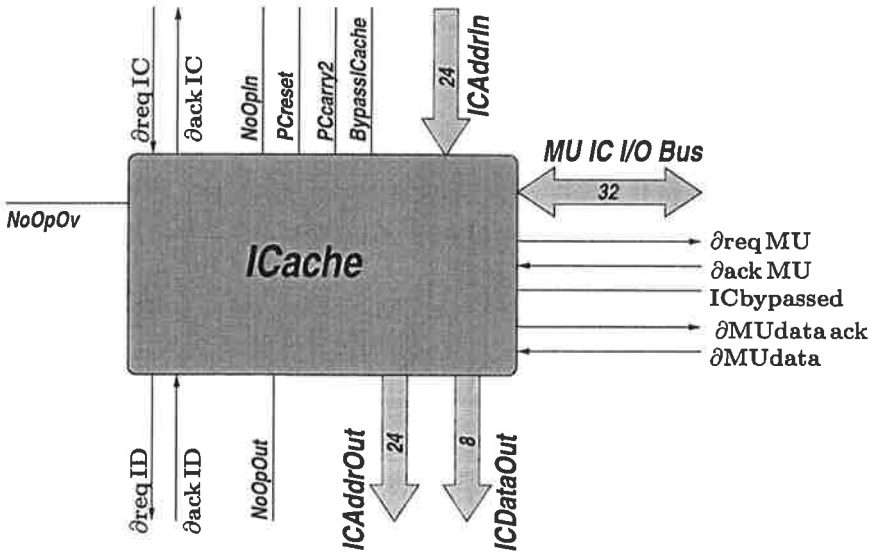


Figure 5.2 ICache Box. The unit interfaces with the Program Counter (PC) and Instruction Decode (ID) units, and interacts with the Memory Unit (MU) during a cache miss.

the ICache and DCache lines). The signal *ICbypassed* indicates that the ICache is operating in external-fetch-only mode, and causes the MU to fetch only one word from memory.

An internal block diagram of the ICache is shown in Figure 5.3.

The parameters of the ICache were investigated using a address generation model that modelled spatial and temporal locality in programs in a semi-random manner [AML94a], as no machine model or compiler for this ISA existed. This suggested that a 2-way set-associative cache with 32-byte blocks was optimal tradeoff between complexity and performance, based on our preliminary floorplan that allocated an ICache size of $2kB$. The SRAM cell aspect ratio gave an array 128 cells wide by 128 deep for this floorplan.

The ISA of Table 5.1 defines a load/store instruction as taking either two or four bytes (requiring either two or four fetches from the ICache to generate a DCache request). Therefore, the ICache is designed in a much more aggressive manner than the DCache, since the frequency of access to the ICache will be much higher than that to the DCache.

2.1.1 Array Architecture

The ICache fetches bytes for the ID unit using an array 128 cells wide, or a total of 16 bytes. This bandwidth would be effectively wasted if the SRAM were accessed on every cycle where the data was available in the last fetch. Therefore, the array latches output values and holds them in the event that the next access falls into this range. Thus, the array holds a small range of sequential data. The *width* of the array is split between the two sets (so that

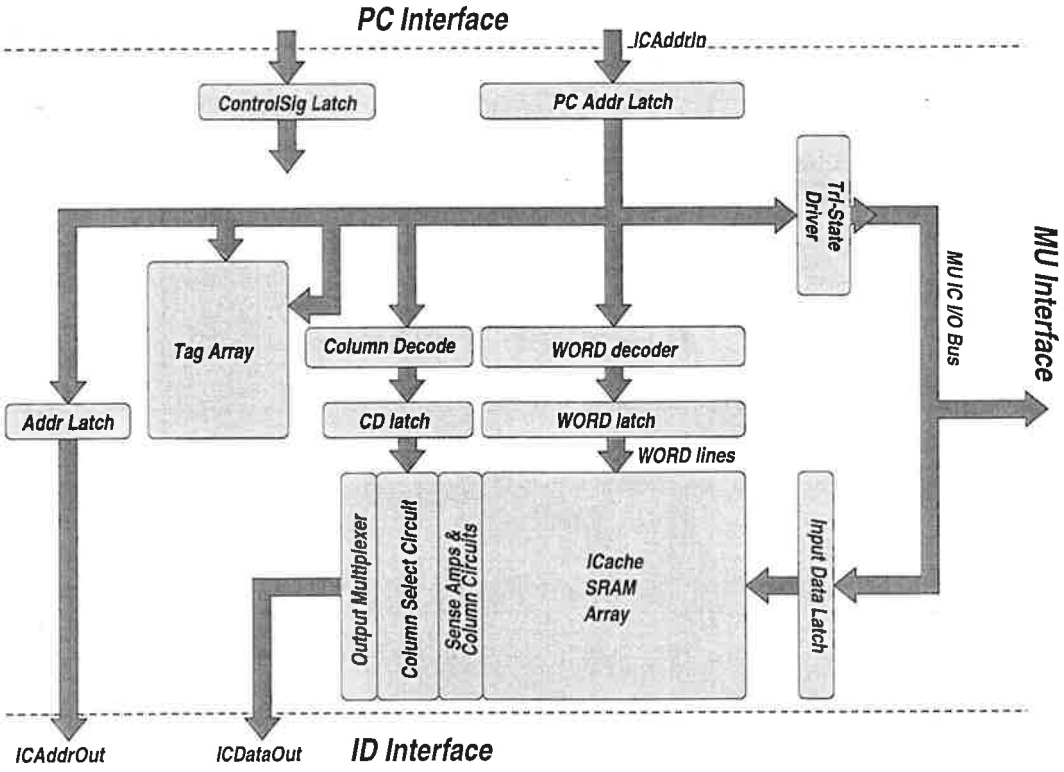


Figure 5.3 Instruction Cache Internal Architecture.

array decode can proceed without prior determination of the required set), and thus eight sequential bytes can be ordered into the one row of the array.

The array architecture is shown in Figure 5.4. Four bytes can be stored from the one fetch in the output latches, and the next four bytes can be stored in the self-timed latched sense amplifier while the array precharges, shown in Figure 5.5. The circuit is sensitive to $BITcol$ and $\overline{BITcol}$ when they drop below V_{tp}, the pMOS device threshold. When the sense amplifier output is desired to be held, the column select multiplexers, which pass data onto the $BITcol$ and $\overline{BITcol}$ lines, are deactivated and the $HoldData$ signal is raised. This holds the state of $BITcol$ and $\overline{BITcol}$ in the current state of the sense amplifier circuit (the off-current leakage of the column select multiplexer should hold the other node at a high level, since the nodes on the other side of the column select multiplexer are precharged). The use of a traditional differential sense amplifier [WE95, Chapter8] was initially considered, but the tight timing requirements on the activation of the amplifier required a solution which allowed the sense amplifier to be "fired and forgotten", as in the case of the sense amplifier of Figure 5.5, in which *EqualiseSense* is taken low before the access begins. Completion data from the SRAM is generated by inspecting the outputs of one of the sense amplifiers, which provides a differential precharged output. These outputs are ORed, which constitutes a completion

signal from the array.

Thus, from one SRAM row access, eight bytes of data are obtained and held for access. The control can take advantage of this and simply change multiplexer control values, skipping the SRAM access when a sequential order is detected.

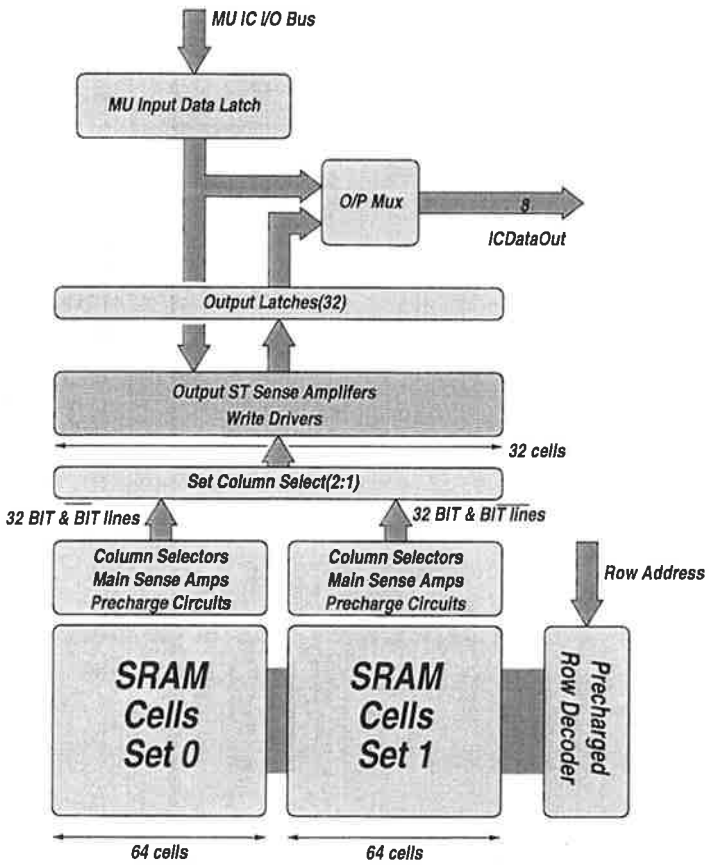


Figure 5.4 ICache SRAM Array Organisation. The physical layout used a modified placement to reduce wiring length and the number of multiplexers signals must pass through.

2.1.2 Cache Tags

The ICache is two-way set-associative, requiring two sets of tags to be stored, accessed and compared for each block address. A number of schemes were explored for the tag comparison [AML94a], but the final design was relatively simple. The tag array and tag column cell architecture are shown in Figure 5.6.

Incoming tag and address data is sent to the column circuit and row decoder, respectively. When evaluation of the tag commences, one of the tag WORD lines goes high, causing a downwards transition on either $BITtag[i]$ or $\overline{BITtag[i]}$. The sense amplifier of Figure 5.6(b)

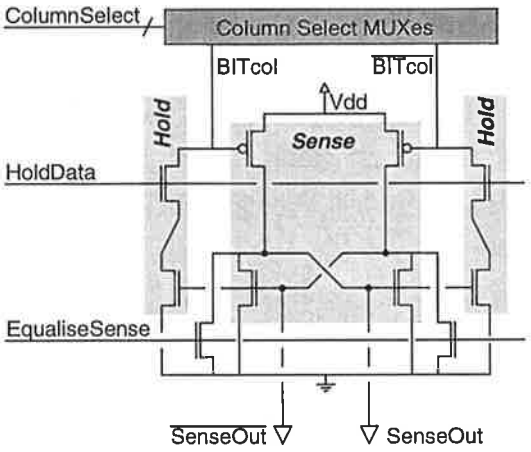
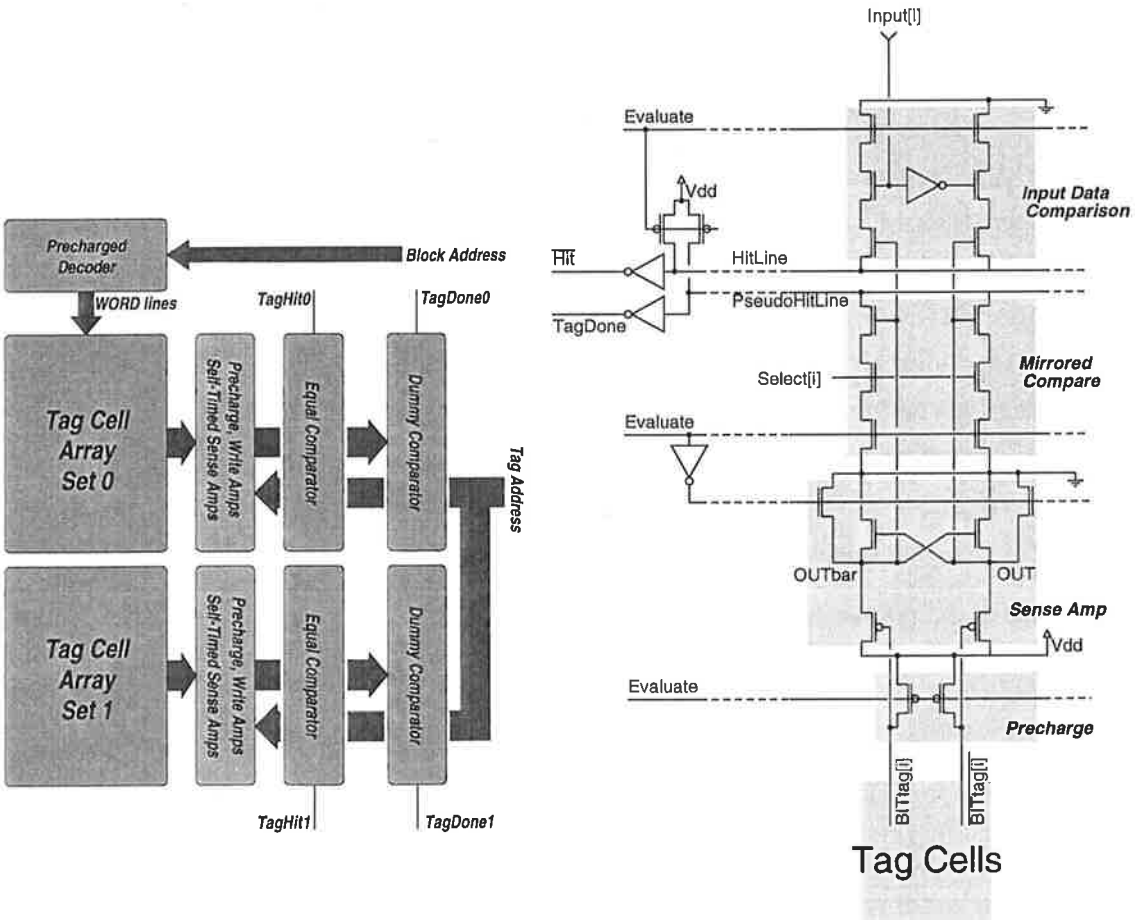


Figure 5.5 Latching Self-Timed Sense Amplifier.



(a) Tag Array Architecture

(b) Tag Unit Column Circuits

Figure 5.6 ICache Tag Circuits. The array has two sets of tags to track the data stored in the SRAM. The sense amplifier used in (b) is similar to that of Figure 5.5, but has the data holding circuits removed. This schematic does not show the write amplifier for the column.

is ideal for self-timing because its outputs are precharged low, and only one output evaluates high during sensing. The sense amplifier outputs drive two comparator circuits. One circuit, the *Input Data Comparison* circuit, pulls down the line *HitLine* if a mismatch between the stored data and the input bit occurs. Thus, any mismatch in the input tag word to the stored cell data causes this line to go low, indicating a miss. The second circuit, the *Mirrored Compare* circuit, has all but one $Select[i]$ low. Thus, if the transistors and wiring in these two circuits are duplicated exactly, the *PseudoHitLine* will pull down in the worst-case comparison time of the real input data, and can thus be used to provide the signal $TagDone$, which indicates that the signal $\overline{Hit}$ is valid.

The tags unit also contains the LRU/Valid unit, which contains LRU (least-recently-used) information for the selection of replacement blocks from the two sets, and information on the validity of each block in the cache.

2.1.3 Pipelining

The expected latency of SRAM row decode was quite high, and this task must be completed before SRAM array access can commence. The ICache was pipelined into two stages to mitigate this latency during SRAM access by placing the row decoder in the first stage, and then latching its output and placing the SRAM array in the second stage. The second stage contains the SRAM array, the row decoder latches and the ID interface, and the first stage contains everything else. This scheme does improve bandwidth during SRAM accesses, but adds latency to the pipeline and complicates the control slightly. It also improves cycle time during sequential accesses as the first stage control can service a new request very shortly after determining a sequential access has occured, handing multiplexing data to the second stage and beginning a new access from the PC.

2.1.4 Control Aspects

The control of the ICache is reasonably complex because of the number of control signals that must be generated. However, some illustrative examples can show how the control system was devised.

Overview

The control schema was planned so that *blocks* of control could be designed to fulfill various functions, and then interconnected to effect the total desired function. An overview of the ICache control schema is shown in Figure 5.7.

Addresses arrive from the Program Counter at the input to the first stage of the ICache. Stage One may detect a cache miss, in which case it activates the *Miss Engine*, which controls the fetching of data from external memory via the Memory Unit (MU), itself interacting with

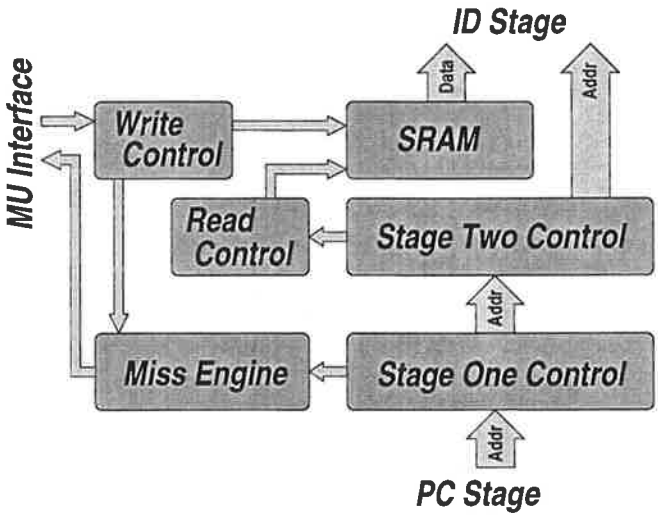


Figure 5.7 ICache Control Overview.

the *Write Control*, sequencing the writing of data into the SRAM. When ready, Stage One issues to Stage Two, which controls the read operation of the ICache SRAM. If the Stage Two logic detects that a new fetch is required, it activates the *Read Control* block, and when data is ready, Stage Two issues to the Instruction Decoder (ID) stage.

First Stage Control

A schematic overview of the control of the first stage is shown in Figure 5.8. The input S-Pipe stage latches input data, and begins some initial computations that determine whether the SRAM access can be skipped (when within the range of the output latches of Figure 5.4), the tag access can be skipped (when the address is still within the same 32-byte block), or a complete new evaluation must occur. These computations, and any subsequent tag access required, are triggered by the signal $PreActiveS1$. When this has completed, the signal $GoToActive$ goes high (but only when $HaltICacheStageOne$ is low, a feature added to support unit scan), initiating any further actions necessary (like row decode or miss processing). All Stage One actions have completed when the signal $GoToStage2$ goes high, sending data to Stage Two for further processing.

Line fill is initiated when a tag miss occurs, and is triggered by the event $\partial process\,miss$, generated by the *feed* gate of Figure 5.8.

Line Fill

The line fill circuitry sequences new addresses and new address decoding for the row decoder, as well as controlling the interface between the ICache SRAM write controller and the Memory Unit (MU) outputs. The line fill circuit consists of two loosely coupled controllers, the *miss*

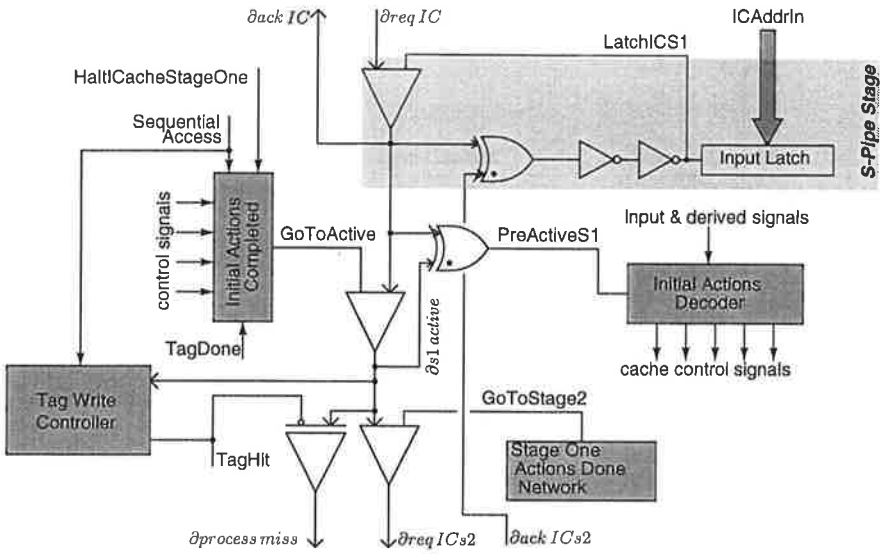


Figure 5.8 ICache Stage One Control.

sequencer and the separate *SRAM write controller*.

The *miss sequencer* controls the addresses and the evaluation of the row decoder, and indicates to the Stage One control when miss processing has been completed. A schematic of the control is shown in Figure 5.9. The line fill circuit also stalls until the second stage is free. This ensures that the second stage is ready to receive data when it arrives from the MU, and so that a potential deadlock condition in the pipeline is avoided (if the pipeline is full, the DCache and ICache require access to the MU at the same time, and the ICache gains access first with the second stage busy, the pipeline would deadlock).

The miss sequence is triggered by the $\partial process\ miss$ signal, generated in Stage One. A small counter which increments the row and column address data for the 4-byte per access line fill must be loaded and sourced as input for the row and column decoders. Concurrently, the Stage One ICache address is driven onto the Memory Unit I/O bus for the ICache, controlled by the signal $DriveICaddressToMU$. When all preconditions for the start of line fill have been completed, the line fill *loop* begins. When the write operation has commenced in the SRAM (this is controlled by the write control circuit, operating directly off the MU output signals), provided there are still more accesses required, the loop counter is incremented and the row decoder re-evaluated. Once the SRAM write completes, this loop begins again.

The second component of the line fill circuit, effectively residing in Stage Two, is the control for the write of the SRAM. A schematic of the control structure is shown in Figure 5.10.

An arriving data value from the MU (via $\partial MUdata$) causes the incoming data on the

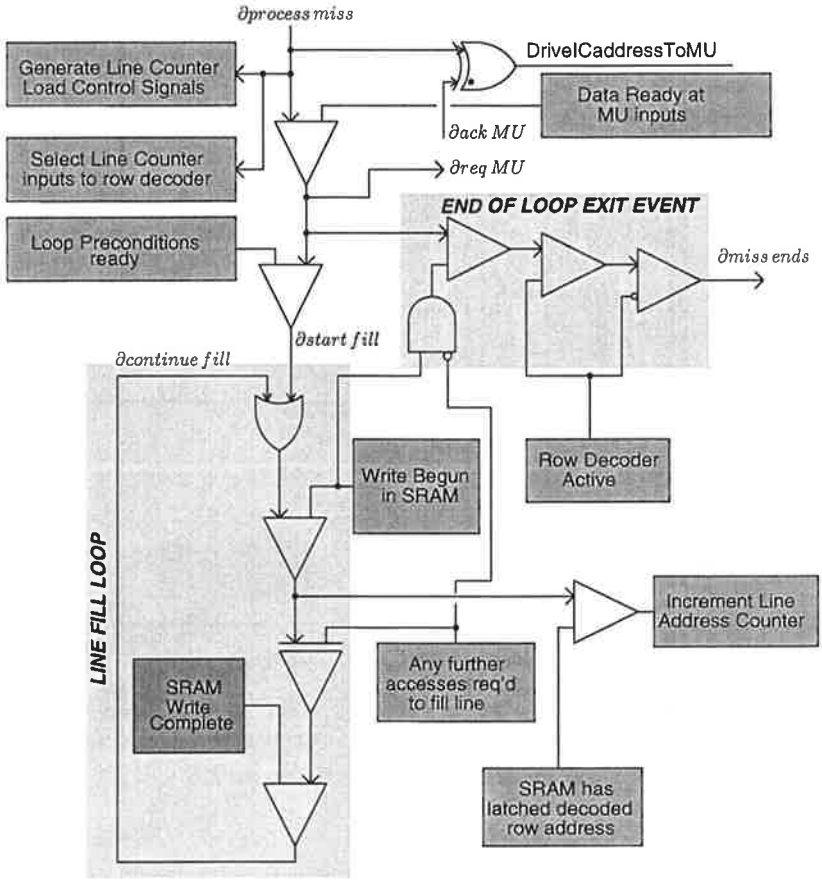


Figure 5.9 Miss Sequencing Engine. The miss engine asserts control over the ICache row decoder to setup addresses correctly for a line fill operation.

$MU\,IC\,I/O\,bus$ to be latched, and the array precharge function to be deactivated. The latching circuit uses a modified S-Pipe. The *until* gate is connected directly to $\partial MUdata$, with the control design assuming that the time between arriving values is greater than the SRAM write time.

Once data is latched, the column write amplifiers are enabled, which swing the relevant BIT and $\overline{BIT}$ lines in the array. When this completes, the signal $ReadyToWrite$ goes high, enabling the latching of the row decoder — one of the array WORD lines will now enable, writing any cell whose BIT and $\overline{BIT}$ lines have been set (any cells' lines which have not been set will be precharged, allowing the cell to swing the lines as if it were a read cycle, nondestructively). The completion of this action, signified by $RowDecoderLatched$, causes the row decoder outputs to be set low (thus deactivating the cell write), followed by the re-enabling of array precharge (when $DisableArrayPrecharge$ goes low). The signal $WriteIsActive$ is fed back to the *miss engine* of Figure 5.9 to indicate the status of the SRAM write.

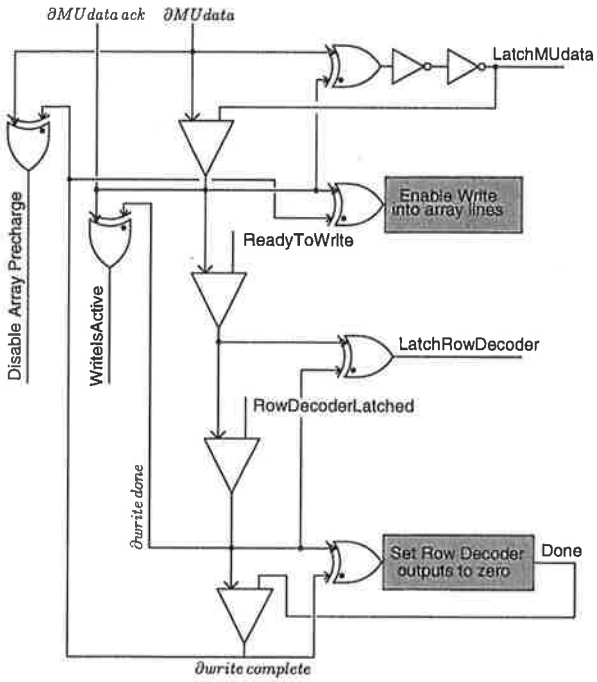


Figure 5.10 ICache SRAM Write Controller. The write controller sequences the required array operations, directly triggered by the data output signals sent from the MU.

Second Stage Control

The second stage of the ICache is an autonomous unit that handles accesses associated with reading the ICache SRAM. A schematic of the ICache stage two control is shown in Figure 5.11.

The basic input section of Stage Two is again a S-Pipe, however, there is an additional *send* gate in the return acknowledge path to ensure the row decoder has been properly latched (only when it needs to be latched, as controlled by the OR gate enabling the second *send* gate). When Stage Two is ready to issue data to the Instruction Decode stage, the signal *S2ReadyToIssue* is raised (a halt signal is added here to facilitate Stage Two unit scan). An incoming request from Stage One also supplies the signal *FetchFromSRAM* when a new fetch cycle is required. The read operation of the SRAM is not required on every second stage request, but is detected when required and the signal $\partial fetch\ SRAM$ issued. This feeds into the read control structure, shown in Figure 5.12.

The read control sequences the reading of two parts of the SRAM which are accessed on the same row. The arriving signal $\partial fetch\ SRAM$ enables the main sense amplifiers, which swing every BIT and $\overline{BIT}$ line in the selected set (the control structure of Figure 5.11 ensures that the latching of the row decoder has completed by this time). The internal precharge of the column sense amplifiers is then deactivated, followed by the read of the low set of four bytes

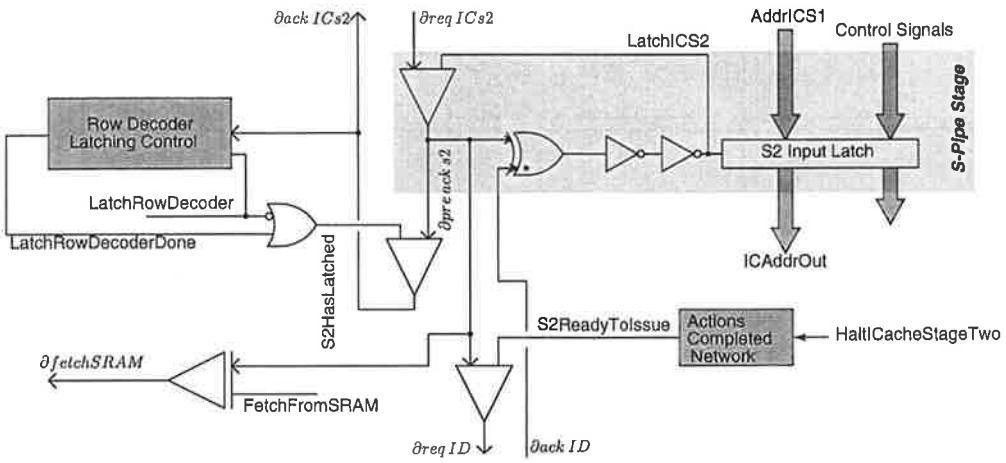


Figure 5.11 ICache Stage Two Control. Stage Two completes the SRAM access, where required, by accessing the array. Otherwise, it delays the setting of the signal *S2ReadyToIssue* until the array column multiplexers have had time to switch the output byte to the new value.

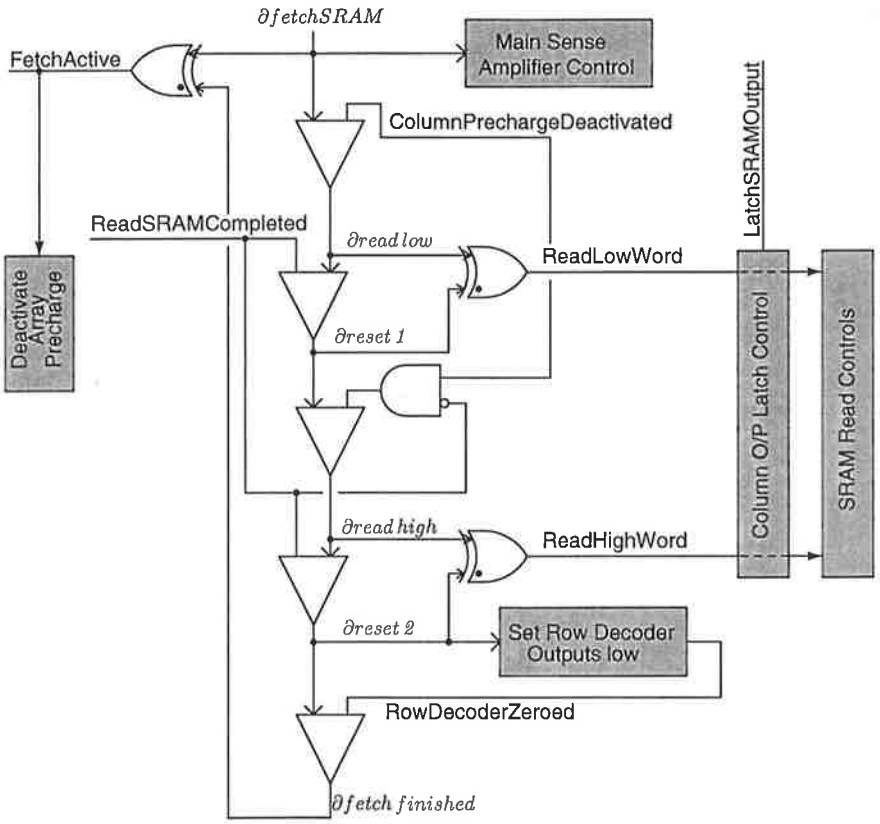


Figure 5.12 ICache SRAM Read Control. The trigger event, *∂fetch SRAM*, is supplied by the Stage Two control structure of Figure 5.11.

referred to by this address. The column sense amplifiers are then recharged, and the high set of four bytes are read. Control circuitry ensures that the correct four bytes ends up in the column output latches at the end of the read cycle. Once the two reads complete, the row decoder outputs are set low (or *zeroed*), and the array and column amplifiers are precharged in readiness for the next cycle.

2.1.5 Verification

Major portions of the ICache control were simulated using VHDL, with appropriate ECS gate models and some initial timing data inserted. This validated that the control operated as expected. The design then proceeded to layout. At this stage, major portions of the ICache control were verified for timing and functionality by HSPICE [Met96] simulation. Some control fixes were required to ensure timing was met. These portions were then interconnected and simulated, using the switch-level simulator IRSIM [Adv95a] for functional testing.

The verification that the ICache was performing as expected was based around custom designed random unit *exercisers*, that were developed to run the ICache exhaustively through functional tests using the IRSIM simulator. The ICache SRAM was validated overall by a custom exerciser that checked every memory location in every possible access mode, using random data. This involves two passes of the simulation. The first applies the input data and monitors the simulation output for the relevant completion signal from the array, and stores these completion times in a data structure. Checks on the values of the output, called *assertions* by the simulator, are then added to the input file and the second simulation pass is then performed. When applied to the ICache SRAM, several errors relating to bypass handling and sequential read accesses were uncovered and corrected.

The complete ICache, including SRAM array, was then generated, and another custom unit exerciser developed to randomly test its functionality. This program exercised every ICache location in every access mode, and validated its functionality.

2.1.6 Performance

Performance figures for the ICache, shown in Table 5.2, are based on simulations using IRSIM. IRSIM is an accurate simulator for gate networks (it was *tuned* to correlate with a Level-3 MOS model, and gave good results), but the analog circuit effects of the SRAM array, tag circuitry and some feedback nodes cause some problems for IRSIM which may decrease confidence levels in the simulation results. The SRAM access time is approximately $12ns$ from simulation — the ICache access takes longer because it first determines whether an ICache *hit* occurs before commencing SRAM access. Even with this penalty, the average access time of the cache is low.

Operation	Hit-Seq4	Hit-Seq8	Hit-Block	Hit-Cache	Miss-Cache
Cycle Time	5.7ns	5.3ns	26.7ns	28.8ns	30.6ns+T_{ml}
Stage One Latency	3.1ns	2.8ns	7.8ns	11.3ns	26.9ns+T_{ml}
Stage Two Latency	3.0ns	3.3ns	25.3ns	27.4ns	27.5ns
Avg. Cycle	8.3ns				

Table 5.2 ICache Performance Figures. These results are based on IRSIM simulation, which does not model analog circuit effects particularly well. T_{ml} is the miss latency, the time from *∂reqMU* to the last datum being received from the Memory Unit. The *Hit-SeqX* parameters relate to the time taken for the ICache control to determine that a sequential mode exists, and simply change multiplexer values to produce data.

2.2 Data Cache

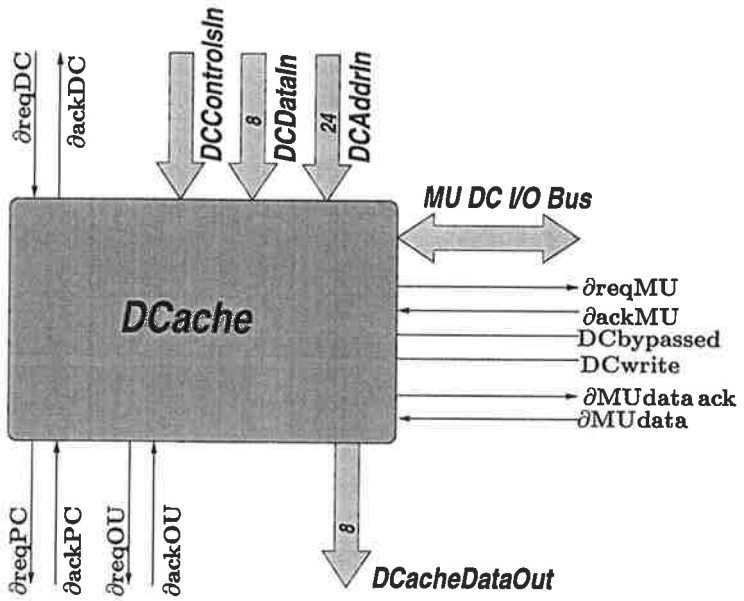
An overview of the DCache structure is shown in Figure 5.13. The DCache interfaces with the ACS, which provides *ACSAddress*, *ACSData*, and some control signals for various other functions. There are three interfaces with other units. The DCache can send data to either the Program Counter unit (via the signals *∂reqPC* and *∂ackPC*) or to the Order Unit (via the signals *∂reqOU* and *∂ackOU*). The DCache also interfaces with the Memory Unit for stores and line fills during a cache load miss.

The parameters of the DCache were investigated using a spatial and temporal access model that was expected to mirror how real data accesses behaved [AML94a]. These simulations were not conclusive, but suggested a line size from 8 to 32 bytes was appropriate, with a two-way set associative being preferable to a direct-mapped organisation. The line size was chosen as 16 bytes, which with a 2-way set-associative cache of $1kB$ in size (from floorplanning considerations), gives 32 blocks per set. This gives the same parameters as the ICache, allowing re-use of the ICache Tag and LV units with very little modification, and avoiding verification of a new unit.

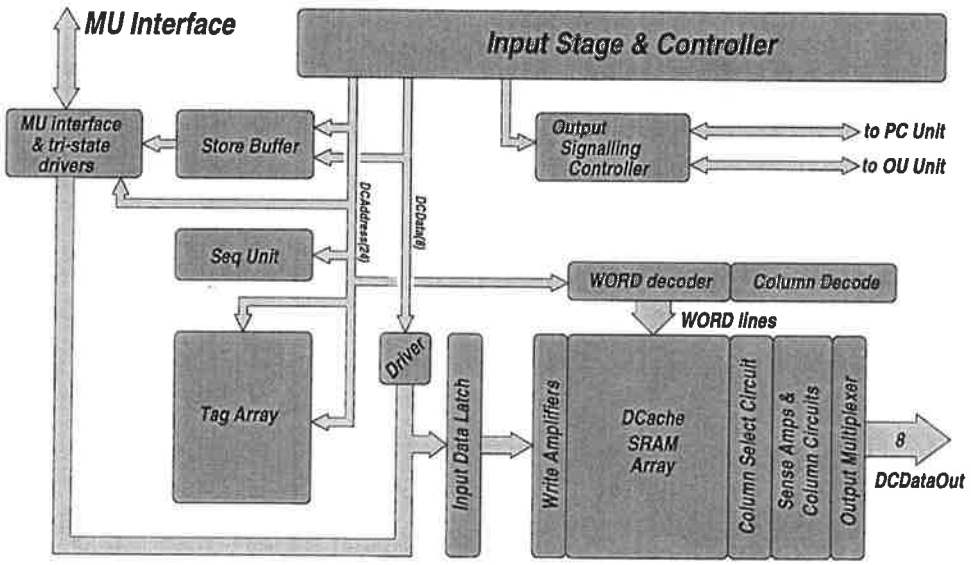
The internal block architecture of the DCache is shown in Figure 5.13(b). The *Seq* unit is a special unit that stores the last upper portion of the read address, and compares it against the incoming address so that the access of the SRAM can be skipped when possible.

2.2.1 Array Architecture and Tags

The SRAM array for the DCache is quite similar to that of the ICache, shown in Figure 5.4. The DCache is smaller, and holds 4 bytes at a time in the output latches, thus there is no need for the latching sense amplifier of Figure 5.5. The DCache write input can come from two sources — the line fill data from the Memory Unit, or from *DCData* for a write-hit. This



(a) DCache Box



(b) DCache Internal Structure

Figure 5.13 DCache Overview. The DCache receives data from the ACS stage, and sends data to either the PC or OU when a cache load is requested. The DCache interacts with the MU for external memory operations.

requires a controlled bus and some write circuit controls to be added to the array structure.

The DCache Tag block is exactly identical to the ICache unit (but has column wider by one bit because of the smaller DCache block size), saving time on the design and verification of a new block for the DCache. An additional unit to determine if a sequential address has been requested, the *Seq* unit, sits above the DCache. It is structured similarly to the comparison that occurs in the tag block, and thus the *Seq* unit is assumed to have completed by the time the tag unit completes.

2.2.2 Store Buffer and MU Interface

The DCache incorporates a small store buffer to hold stores. The cache is write-through, and thus the core would stall completely when doing a sequence of STORE instructions (for example, a CALL instruction) as the DCache would stall after the first STORE while waiting to write it to memory. The addition of a three-entry store buffer was designed to stop the core stalling under such circumstances while fitting in with the available floorplan.

The block architecture of the store buffer and Memory Unit interface is shown in Figure 5.14. It consists of a three-stage S-Pipe to hold incoming store requests, and an interface circuit for sharing the port to the MU between the store buffer and a load miss request in the DCache.

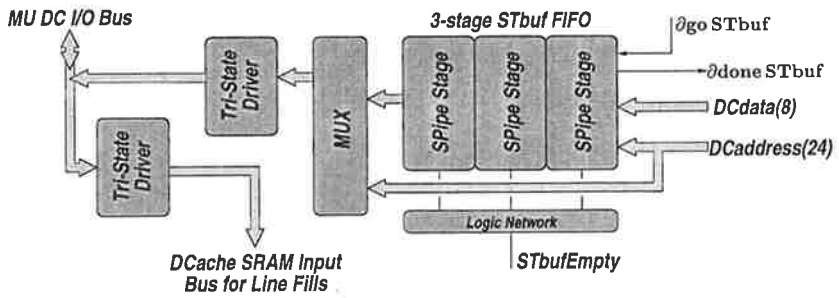


Figure 5.14 DCache Store Buffer (STbuf) and MU interface.

2.2.3 Control Aspects

The control of the DCache is complex due to the large number of operations that the DCache is required to handle. The DCache handles LOADs (hit, miss or bypassed), STOREs (hit or miss), enabled cache flags (for both ICache and DCache) set by the ICDS/EN and DCDS/EN instructions, and the FLSH instruction, which clears the DCache tags.

Overview

The structure of the DCache control is shown in Figure 5.15.

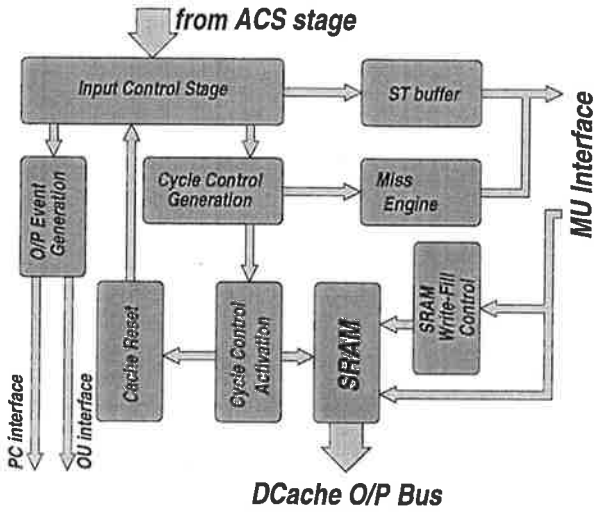


Figure 5.15 DCache Control Overview.

The input stage receives data out of the ACS stage, latches it, and then commences some initial actions, including tag comparison. Once the result of the compare is known, the *Cycle Control Generation* determines the required operation type and initiates it in the *Cycle Control Activation* block. This sequences any actions required in the DCache, and also provides the completion signal to the DCache input stage via the *Cache Reset* block. The Store Buffer and Miss Engine both share the port to the Memory Unit, which requires an access strategy (detailed below). The returning data from the MU goes to the DCache SRAM, with the SRAM Write-Fill Controller sequencing the actual DCache write during Load Miss cycles.

Input Stage

A schematic overview of the DCache input stage is shown in Figure 5.16. The input S-Pipe stage latches input data, and then activates the signal *DCachePreActive*, which enables initial actions including tag comparison. Once the tag has completed (and the cache is not halted, a testability feature to enable unit scan), the unit goes to the *DCacheActive* state, which enables any relevant actions to occur as dictated by the type of cycle detected. Once these actions have completed, the signal $DCfinished$ is raised to provide an internal ∂ack event for the DCache.

Store Buffer and MU Interfacing

The Store Buffer, which is essentially an autonomous unit, contends with the DCache Load Miss mechanism for access to the MU port. If the design allowed the miss to get data ahead of the completion of the stores outstanding in the buffer, it would create a potential consistency problem [PH96] with the miss data. Therefore, any load miss *stalls* until the store buffer

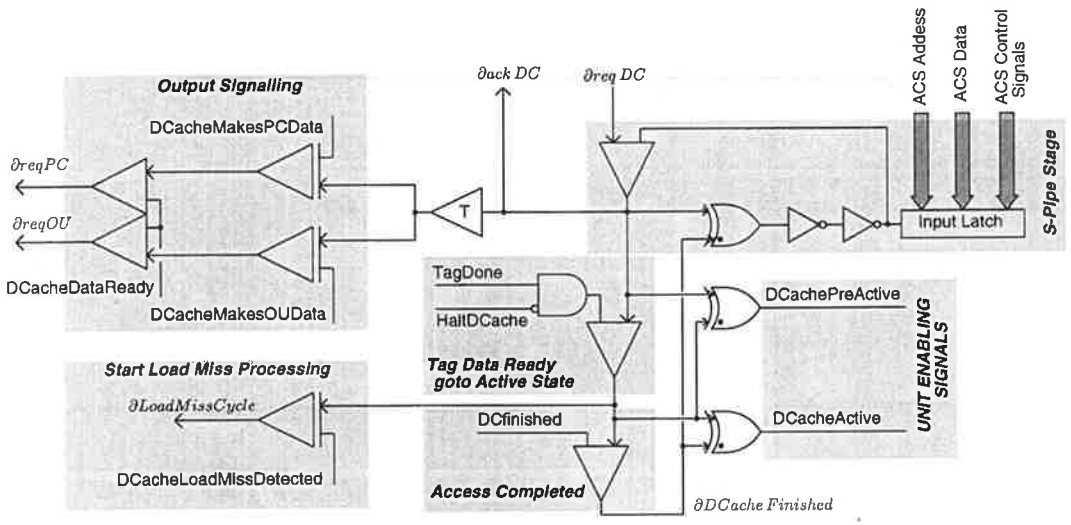


Figure 5.16 DCache Input Stage and Block Control Signals.

is empty. This also eliminates any contention resolution issue on access to the port. The structure of the Store Buffer interface to the Memory Unit is shown in Figure 5.17.

The *miss engine* and other parts of the control of the DCache are similar to that used for the ICache [AML95].

2.2.4 Verification

A VHDL model of the DCache control was used to verify that the logic operated correctly. The initial control layouts were simulated for function and timing using HSPICE, and then interconnected and simulated using IRSIM. Once the design was operational and all timing issues were being met, testing moved into the verification stage.

The verification of the DCache proceeded in a similar manner to that used for the ICache. The DCache SRAM was verified by using a custom *unit exerciser* which called the simulator IRSIM in two passes to determine that the unit was operating correctly. Another unit exerciser ran exhaustive tests on the DCache, using every possible operation type with random data. After the correction of a few routing faults, this program validated that the DCache was operating as expected.

2.2.5 Performance

The performance of the DCache was determined for each access type using IRSIM simulation. Again, the IRSIM simulator does not cope well with sophisticated analog circuits, causing some problems with the simulation of the DCache and the interpretation of the results. The goal at saving DCache power was successful, but the DCache read access time is too long, at 30.6ns compared to the array access time of around 12ns (from HSPICE simulation). This

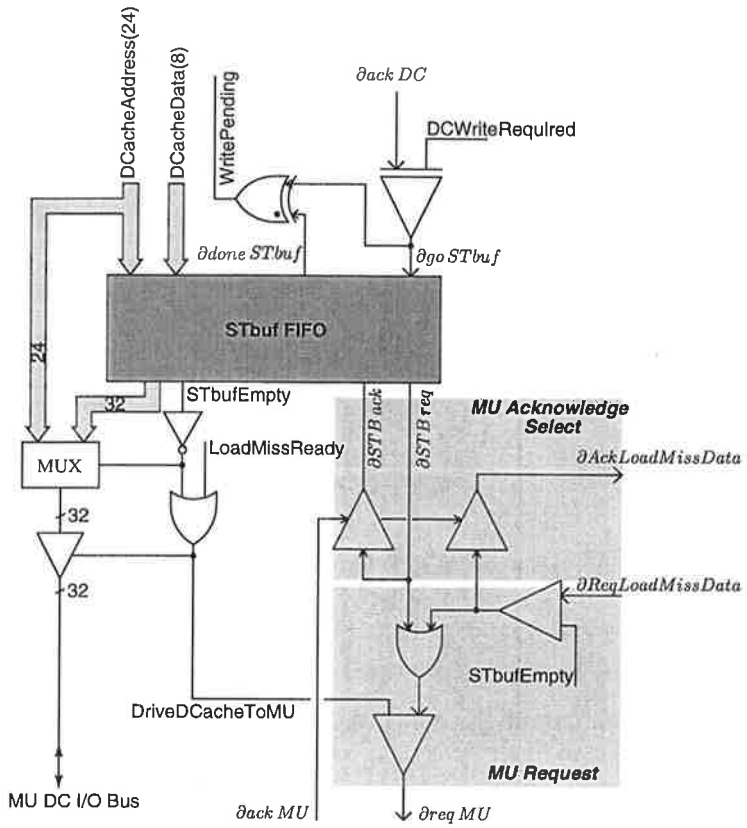


Figure 5.17 STbuf and MU interface control.

is because the *tag* and *seq* units are checked before access commences on a hit, which adds to the critical path.

Operation	LD-Hit-Seq	LD-Hit	LD-Miss	ST-hit	ST-miss	Misc.
Cycle Time	12.9ns	30.6ns	45.5ns+T_{ml}	17.8ns	14.3ns	12.0ns
Latency	9.9ns	25.4ns	39.8ns+T_{ml}	–	–	–

Table 5.3 DCache Performance Figures. T_{ml} is the miss latency, being the time from *∂reqMU* to the last word from the Memory Unit being received. The *LD-Hit-Seq* access occurs when there are consecutive requests to four aligned bytes of memory.

2.3 Memory Unit

The Memory Unit provides the interface between the two on-chip caches and the external memory system. A structural overview of the MU is shown in Figure 5.18. The MU consists of a simple datapath whose function is to latch one of two addresses arriving from the cache units, drive it to the pads, return any incoming data to the relevant unit, and, in the case

of a load, increment the address until the complete fetch is completed. The control block is divided into a number of modules. The *Input Controls* block receives the incoming requests for access to the memory system from the caches, and interacts with the *Arbiter* to resolve contention issues. Once arbitration and latching are complete, the access is setup in the *Startup Control* module, which also interacts with the *Synchroniser* module which asserts control over the external synchronous bus according to the bus protocol. The *Loop Control* is begun for all accesses from this point, even those requiring only a single transaction (like a memory write), which determines the number of iterations to perform. This block feeds out to the *Output Event Control* module, which returns events to the relevant cache when data arrives at the MU during a memory read operation.

External Bus Protocol

The external bus is synchronous and is designed to support commodity memory devices and provide access to I/O while the CPU is active [AMJL94]. The CPU asserts control over the bus in the high phase of the external clock by pulling down $BusBusy$, a precharged external signal. The CPU data is driven out on the next low phase of the clock. If $BusBusy$ is low, the bus is active, and a new request must wait until $BusBusy$ returns high. Memory devices indicate completion by taking the signal $MemoryReady$ low in the low phase of the clock, and then high again after the next $\triangle CLOCK$. The CPU may continue to request data by keeping $BusBusy$ low and driving a new address onto the bus in the next clock low phase.

The Memory Unit, implementing the ECSTAC interface to this bus, keeps $BusBusy$ low during an entire line fill, but resets it after each access.

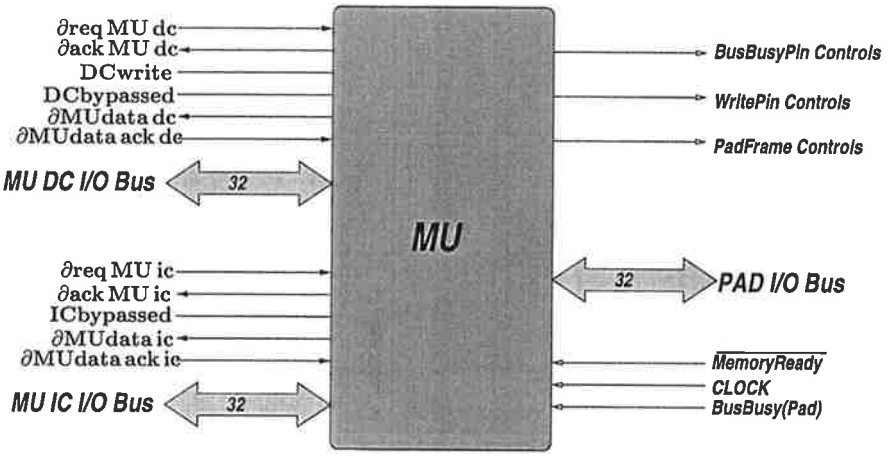
2.3.1 Control Aspects

The two most critical parts of the MU are the input *arbiter*, providing contention control between the ICache and DCache requests to the MU, and the *bus synchroniser*, interfacing between the asynchronous system on-chip and the synchronous bus used externally.

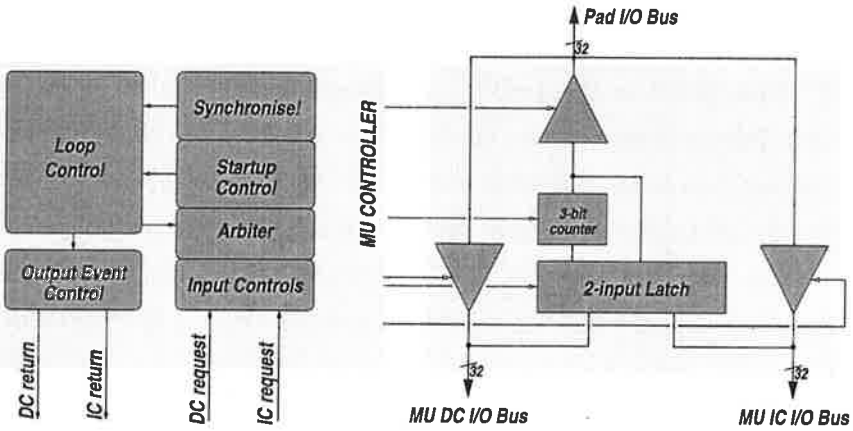
Input Arbiter

The input stage and arbiter determine which unit gets access to the MU resource based on request signals to the MU. As these signals are asynchronous, an arbiter is required to make a deterministic decision, although this can take a potentially unbounded time. This input circuit is shown in Figure 5.19.

Input requests pass through to a circuit which enables $ReqIC$, $ReqDC$, or both. These two signals then pass into an *arbiter* [Pav94, Mar90a, Sei80], which ensures that the signals $GoIC$ and $GoDC$ are mutually exclusive at all times, regardless of the input behaviour and timing.



(a) MU Box



(b) MU Internal Structure

Figure 5.18 MU Overview. The external signals of the MU are shown in (a), while (b) shows the internal datapath and control modules.

When access to the port has been granted, the control loop (see Figure 5.21) is activated by the enabled event $\partial readyMU\ ic$ or $\partial readyMU\ dc$. This enables latching of the input data for the relevant port, which, when completed, allows the associated $\partial ackMU$ event to be returned (either $\partial ackMU\ ic$ or $\partial ackMU\ dc$). When the MU completes the access, the signal $\partial resetMU$ is generated, lowering the currently-active Req signal, and allowing the arbiter to make a new determination of enabled request.

Bus Synchroniser

The bus synchroniser controls the access of the asynchronous MU unit to the synchronous external bus. Synchronisation is needed once *per access*, since the MU can hold control

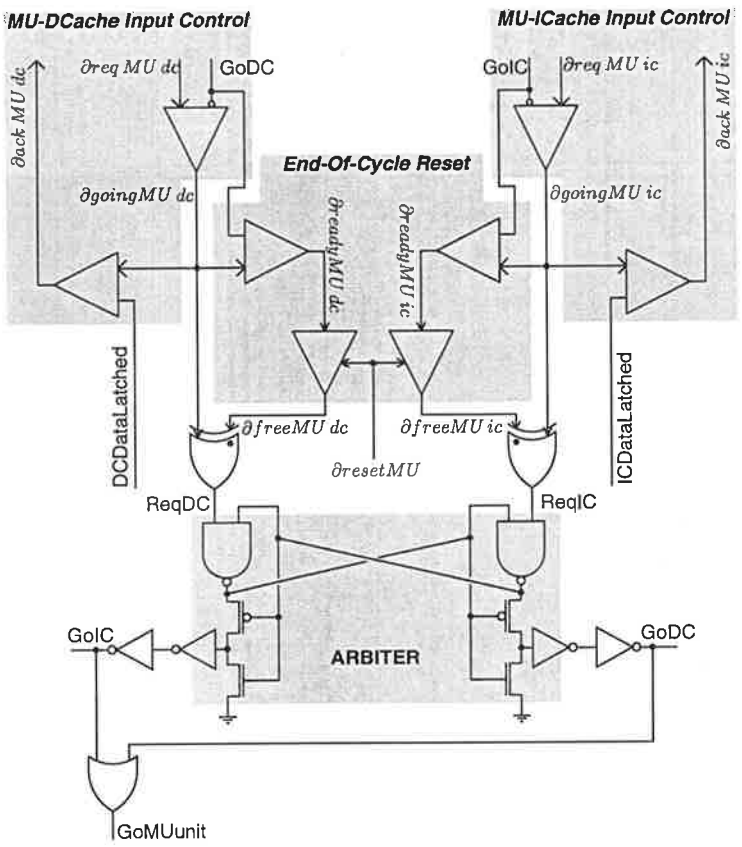


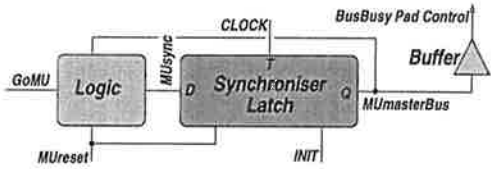
Figure 5.19 MU Input Circuit and Arbiter. The arbiter circuit is a well-known filtered-NAND circuit [Pav94].

of the bus to do a series of memory accesses. Synchronisation has the potential to cause metastability on the output, a well-known and studied problem in synchroniser design [CM73, Sak88]. This problem is unavoidable in digital design [Mar81].

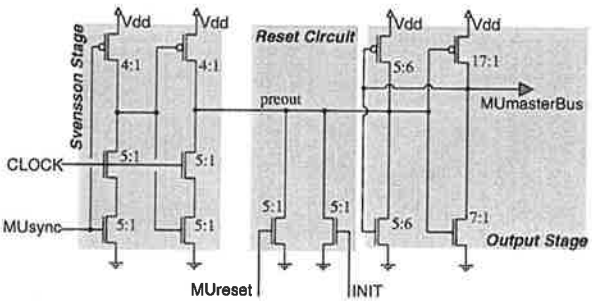
The synchroniser structure and circuit are shown in Figure 5.20. The synchroniser attempts to assert control of the external bus when $GoMU$ goes high (see Figure 5.19). The bus protocol defines the CPU as being able to assert control of the external bus when CLOCK is high, thus, a level-sensitive latch is used as a synchroniser. The synchroniser is optimised for metastability resilience, but Figure 5.20(c) shows synchroniser failure for a narrow range of $DATA \rightarrow CLOCK$ timings of about 5-10ps for the synchroniser circuit used. The synchroniser output stage FET sizing was based on guidelines developed in previous work [AML94b, Sak88], limited by the available pitch of the synchroniser cell.

MU controller

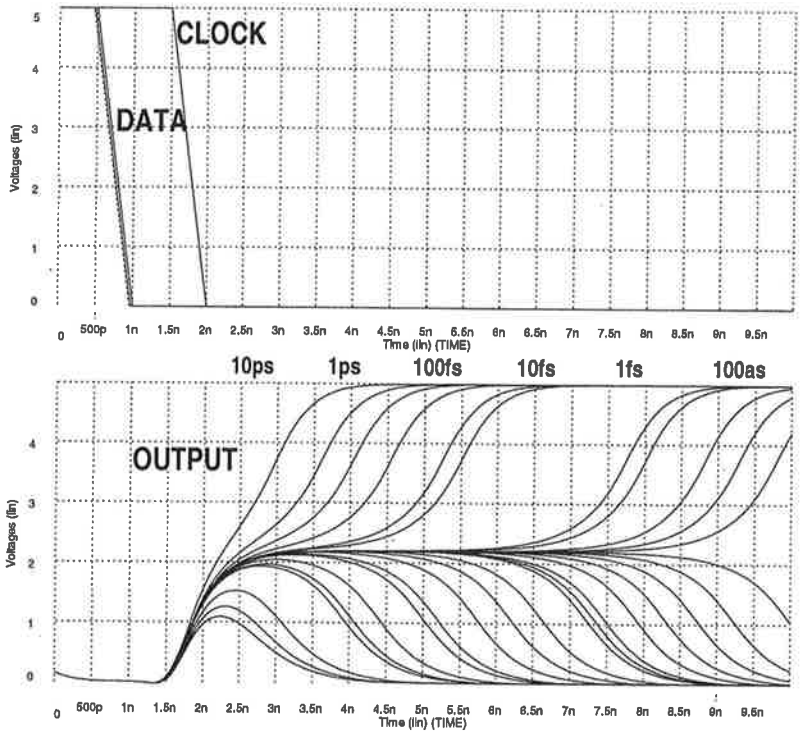
The MU control structure is summarised in Figure 5.21. The loop commences when data has been latched and bus synchronisation successful. The loop continues while there are more



(a) Synchroniser Structure



(b) Synchroniser Circuit



(c) Synchroniser Failure

Figure 5.20 MU Synchroniser. The synchroniser, (a), operates from signals generated internally in the MU, generating the control for *BusBusy*. The circuit used, (b), is a level-sensitive latch [SY92] modified for metastability resilience. The results of a fine HSPICE simulation of the 0.8μm CMOS layout of the synchroniser are shown in (c), with the times indicated showing the level of timing resolution needed to expose failures of the length shown.

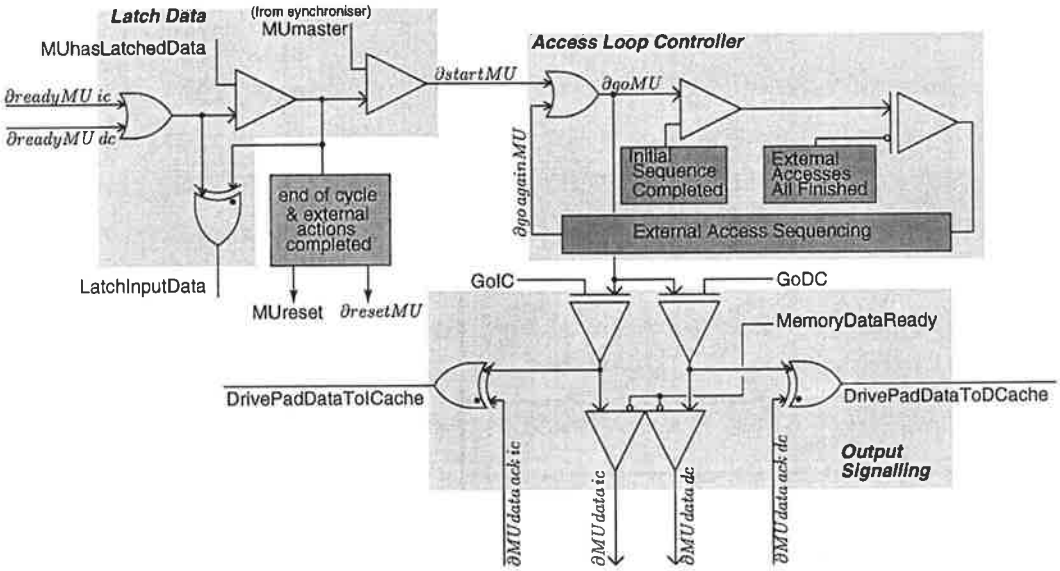


Figure 5.21 MU controller.

accesses required by the requesting port (the MU maintains an internal counter that mirrors the status of the counter in each cache during access), implementing various synchronisation and control actions as the loop progresses. The *Output Signalling* block generates return events to the enabled cache on each iteration of the loop. The event $\partial goMU$ triggers one of the two *feed* gates, and the generated event propagates to the return event $\partial MU data\, ic$ or $\partial MU data\, dc$ when data is returned by the memory system. Signals are also enabled which drive the pad I/O bus back to the relevant unit (and thus the unit receives external data). These signals go active the cycle *after* the request address is given out – this means that extra margin is available on the internal chip buses as the $\partial MU data$ requests only occur after $\bigtriangledown MemoryDataReady$, although internal switching on these buses causes some additional power dissipation.

The access is completed when the block triggered off $\partial startMU$ detects that the access has ended, generating $\partial resetMU$ to reset the event control, and pulsing $MUreset$ to clear the synchroniser.

2.3.2 Verification

The MU is small enough (3150 FETs) to be simulated entirely using HSPICE, using added capacitance to model the effects of the interconnect between the MU and the ICache and DCache. The MU was also tested, using IRSIM, with the padframe added, for proper functionality. The external bus interfacing pads (which drive the BusBusy and signal I/O lines) were custom designed for functionality and to bring padframe area and switching times to be within an acceptable bound.

2.3.3 Chip Verification

The chip verification used the IRSIM simulator exclusively because of the size of the design. Both the ICache and DCache were separately interconnected with the MU, and new custom exercisers implemented to inspect added functionality. The ECSTAC core unit [Mor97] was then added to the MU, ICache and DCache units. At this point, a custom exerciser would have become very difficult and unwieldy, as the memory system outside of the chip would have been involved. Therefore, a number of hand-crafted codes were generated, and a script automatically loads these codes into the ICache memory and runs the simulation. This verified that the unit interfaces were operating as expected, and allowed an infrequent timing problem in the ICache to Instruction Decode interface to be eliminated. This was the only issue encountered in interconnecting the ICache and DCache to the Core components. With the padframe added, the chip was simulated to be performing a sequence of simple instructions correctly.

3 Device Test

ECSTAC was received from fabrication in October 1996. Testing commenced shortly afterwards. A microphotograph of the die is shown in Figure 5.22.

3.1 Initial Test Results

The first tester (V1) for ECSTAC simply verified that the device turned on and did not draw excessive power (two very simple functional and I_{ddq} tests!). A few added components verified that the device was probably requesting address 0 after initialisation (signal noise made this determination difficult). The second tester (V2) verified that the device was indeed asking for address 0 after initialisation. Continued noise problems on this tester (over 1V power/ground noise on this board) required a new implementation.

The third tester (V3) attempted to eliminate the power and ground noise problems with the V2 tester. The use of surface-mount decoupling capacitors and better routing went a long way in meeting this goal. However, the V3 tester revealed a new set of problems.

- Power and ground noise had been significantly cleared up, and is normally very small. However, around the switching intervals noise continued to be excessive. Transient voltages on the chip power and ground pins were around 500mV, corresponding to higher transient levels on the internal chip power buses due to pad inductance [Bak90].
- Signal I/O rise and fall times are somewhere between 2 and 5ns for a heavily loaded pad. This rate of charge/discharge is likely to cause significant problems for pad power

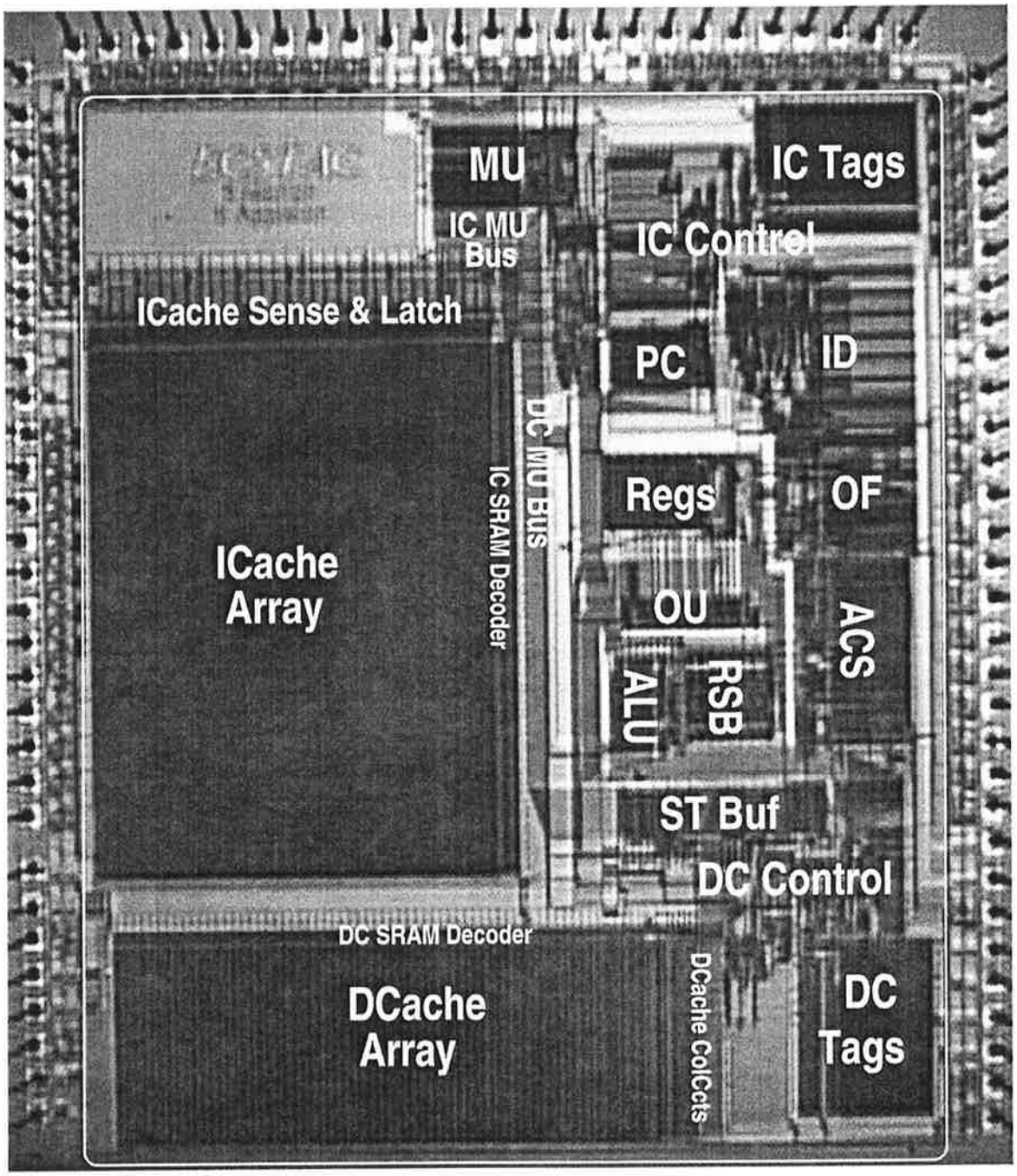
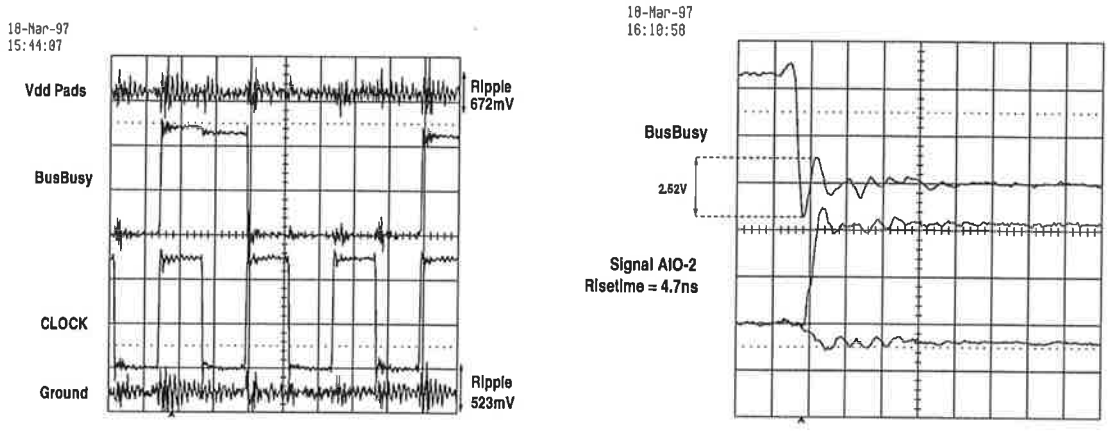


Figure 5.22 ECSTAC chip microphotograph. The die measures approximately 5.0×4.5mm, and was implemented in a 0.7μm L_{eff} CMOS technology [PP94]. The region outside the box is the padframe, supplied from a separate power and ground ring exclusively for pad use.

and ground distribution, which include approximately one power and ground pad per 10 signal pads on the chip.

Oscilliscope traces of the signal noise problems are shown in Figure 5.23.



(a) Pad Power and Ground Supplies

(b) Signal Trace after *BusBusy* transition

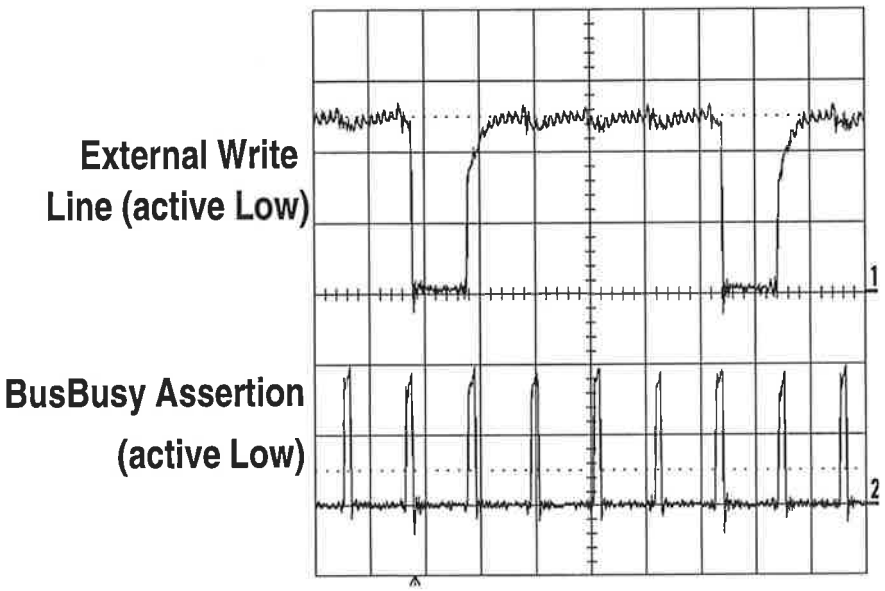
Figure 5.23 Test Jig v.3 CRO Traces. (a) shows the power and ground noise for the pad supplies at the chip pins during switching of the *BusBusy* line, which controls external memory accesses. (b) shows I/O line switching when *BusBusy* goes active. Noise is clearly an issue here, with over 500mV of noise on the *external* pins of Vdd and ground, which will cause significant problems with internal circuits relying on stable Vdd and ground levels.

3.2 Further Test Results

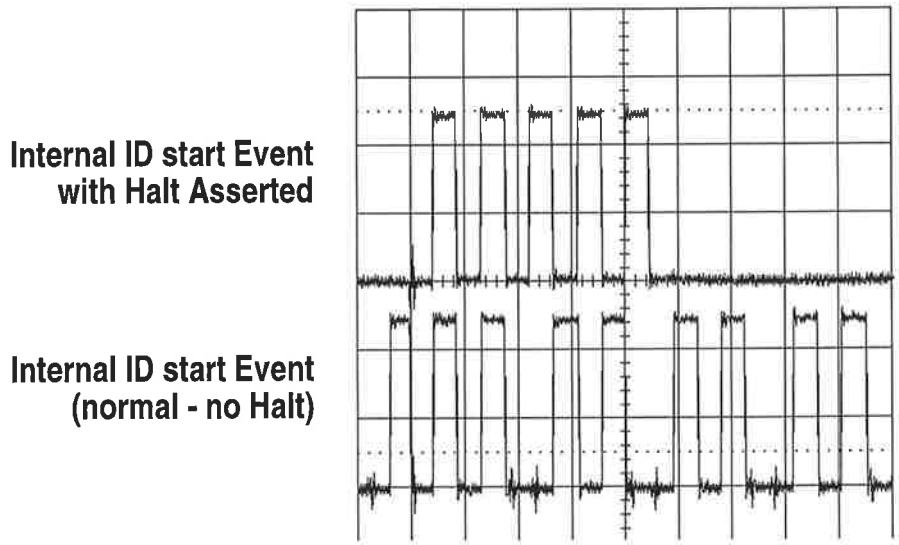
The v4 tester added more features intended to decrease noise levels. All signal I/Os were connected to their traces via series resistors to attempt to decrease peak current drain on the supplies (and thus transient line bounce). The pad power supplies are also fed from buses via small resistors to attempt to damp any transients.

The chip at this point appears to be functional. There is an unidentified problem with the ICache which requires more testing to ascertain the cause of the problem, however the ICache can be bypassed and the chip runs instructions correctly in bypassed mode. Some CRO traces of the chip executing two critical (and observable) instructions are shown in Figure 5.24. The STORE instruction operating as expected is highly promising — an instruction is passing correctly through the entire ECSTAC pipeline, emerging from the DCache and interacting with the MU in the expected fashion.

The continued test of ECSTAC requires more software development — the existing board includes a facility to assert four bytes statically as input, but to go further a more generalised memory system is required. This is the current focus of chip test.



(a) STORE instruction



(b) HALT instructions

Figure 5.24 Operating Trace from ECSTAC. (a) shows a 2-byte STORE instruction being executed after two NOOPs — the write line goes low every 5 cycles, as expected. The ID stage of the pipeline can also be halted using a special instruction, shown in (b). The pipeline otherwise runs continuously until the next initialisation (without a halt asserted), itself very promising because the ICache-MU-ID interaction is therefore functional.

4 Bottlenecks and Challenges

The design, and especially implementation, of ECSTAC raised a number of critical issues concerning future system implementations using ECS. The process of translating a loosely specified architecture all the way to circuits and custom layout proved to be a valuable learning tool as to the utility of the approach.

4.1 ECSTAC Architecture

It is clear that the choice of an 8-bit data, 24-bit address path architecture was a mistake. Our early decision on this point did not consider what the impact of the disparity would be on both control complexity and computation required in the processor core. The combined effect of added control complexity (in the entire machine, but particularly in the OF stage where the variable-length instructions are converted to single instructions), extra complexities in units, the extra unit required (ACS), and the custom ISA all combined to eliminate any imagined benefit of using a smaller datapath width (in terms of area and performance). A full 32-bit machine (performing similar integer operations, with no floating-point or multiply/divide) would have

- been only marginally larger in implementation,
- been much easier to control, with simpler pipeline interfaces,
- and allowed the use of an existing RISC ISA so that proper architecture exploration and chip verification could be done.

The chip area was largely dominated by the two cache SRAM arrays (see Figure 5.22), and thus adding additional area for a larger core would have had only a marginal impact on total die area. Using an existing ISA that has predefined software tools (like compilers and libraries) would allow complete attention to be focussed on the *implementation* of the ISA. Using a custom ISA should be avoided unless it can be justified (in performance or some other metric). There are also architectural lessons to be learned from each of the functional units in the chip.

4.1.1 Main Pipeline

Perhaps the most difficult issue to address in the pipeline is *latency*. Asynchronous designs are highly tolerant of adding latency to both individual stages and units as a whole, because the $\partial req \rightarrow \partial ack$ mechanism decouples timing considerations from the design. Even though some features of the chip may be quite attractive (for example, the ID FIFO, providing ICache access time variation decoupling), the architectural performance impact of these additions was not properly evaluated. Some of these additions will have a severe impact on the real

performance of the machine, especially those that add latency between fetch and execute, such as ICache pipelining, the ID FIFO, and the long latency caused by variable instruction widths arriving at the OF (which has only a single read port to the register file). These cause heavy branch penalties, even more severe in this machine because there is no prediction mechanism.

4.1.2 ICache and DCache

The control architecture of the caches can be characterised as a *check-then-continue* approach — the input data is first analysed, and then a decision based on the result is made. These decisions often lead to activation of long-latency units (the tag unit or row decoder), which subsequently places this initial control on the critical path. Even though this works well for short latency operations (sequential accesses in the ICache), the latency of the unit is then severely impacted when a long latency operation is required. A better approach would have been a *start-check-abort?* approach, starting the evaluation of any units *immediately* (along with the attendant increase in power consumption), and aborting evaluation if the control determines that activating the unit is unnecessary.

In addition, the control method for the access of the array SRAMs (which have a large number of control signals and complex sequences to be issued) needs to be addressed. The access time of these SRAMs is critical, and the present method of generating array controls (see Figures 5.10 and 5.12) is somewhat unsatisfactory, since it places more control on the critical path access time of the device.

The DCache control was complex due to the number of functions it was required to implement. In addition to memory operations, the DCache must handle cache enabled flags and flush operations. This contributes to an expensive control structure. The control method used is *check-then-continue*, and the DCache would be considerably improved if a better matched scheme was used, even though the existing approach saves considerably on power by not evaluating units when they are not required. The usefulness of the *Seq* unit could also be questioned, and would probably not be desired on a 32-bit machine (as the time overhead added by evaluating the unit is almost the same as evaluating the entire SRAM array).

The ICache and DCache can also be *bypassed*. The mechanism by which this was done on ECSTAC compounded the performance issue, since it demanded that the cache controllers handle the situation, contributing to a considerably more complex control flow. The bypassing of these two units was required for two reasons,

- so that the DCache could access I/O regions without causing consistency problems with the memory mapped I/O system, and

- so that the internal performance of the core without caches could be evaluated.

The flags-based mechanism also requires that the DCache store and manage the flags. A better mechanism would be to have externally-applied signals to *kill* any hit signals in the caches (or to perpetually invalidate the cache tags) for performance monitoring, and to use an *uncachable* bit returned from external accesses to invalidate the storage of special data in the caches. This would remove the need for bypass control from the cache logic.

4.1.3 Memory Unit

The MU arbitration and synchronisation mechanisms are adequate because of the long latency involved with memory access. However, the arbitration mechanism of Figure 5.19 does add significant latency to an input request, which could be reduced by using an alternative arbitration scheme. The arbitration mechanism in the MU is safe, but involves a heavy overhead. The time from a request to the acknowledge is around 11 gate delays, provided no contention delays occur, and takes 8 gate delays to assert control of the bus (provided no synchronisation failures occur). A better approach would be to latch data using two latches (one for each of the ICache and DCache buses) and then multiplex the required value after arbitration completes, and also synchronise the bus to the MU as soon as *any* request arrives, rather than waiting for arbitration to complete. This approach is more aggressive (and will probably require that the arbiter is assumed to have completed after a certain time), but will result in higher performance. Even though around 10 gate delays for control functions does seem acceptable, the external memory system *is* a valuable external resource, and every attempt should be made to maximise its utilisation while reigning in latency.

In addition, an improved bus protocol should be employed in future, and more thought directed at delivering high bandwidth to the MU, perhaps without needing constant address delivery and memory completion signalling (via *MemoryDataReady*) by taking advantage of newer memory system architectures [Pri91, Man95]. Even though this control does not get on the critical path, the loop control and sequencing could be improved in a similar vein to the cache control.

Considerable attention could have been given in the MU design to some form of *prefetch* and/or *buffering*, especially once the bus is mastered. It may be seen from Figure 5.22 that the top-left corner of the chip is empty, an area which could have been used for prefetch control and storage had time been available (unfortunately, the late stage at which final chip floorplanning revealed this gap meant that no time was available to implement a prefetch buffer). The architectural impact of this buffer would, of course, have to be evaluated.

4.2 ECSTAC Implementation

The translation of the architecture and block hierarchy to logic and circuit implementation was one of the most interesting aspects of the design.

SRAM and Tag implementation

The array architecture and column circuitry of the SRAM was very elegant and fast. However, the control structure tended to slow the array down because it required a large number of control signals to be asserted for the access in a certain sequence. In future, some method of better controlling the SRAM access cycles should be developed. At the circuit and layout level, the implementation of the SRAM array, the column circuits, and the row decoder is a very difficult, complex and error-prone task. Extreme care should be taken with the designs of large SRAMs (with plenty of sensitivity analysis), and the design considered carefully.

The tag unit circuits saved considerable power and time in the cache design. Past experience [AML95] with the tag array was tending towards a higher power solution, however the final unit requires the assertion of only two control signals, one of which is off the critical path, and is totally internally self-timed allowing extremely fast operation. This improved compare time by 30%, with an 80% reduction in power consumption, over a more exhaustive comparison method.

CAD, CAD, CAD!

Design processes were totally manually driven. There are many places where simple, specialised CAD tools could have assisted in the process of ensuring that asynchronous processes were operating within bounds.

All blocks in the design were manually placed and routed. This works well for datapath blocks, giving optimal area and conforming to a specified floorplan, and is advantageous because the datapath blocks changed very little once completed at the layout level. However, control structures changed enormously from initial specifications. Each time this control was to be simulated, the entire control block had to be redesigned. This procedure was perhaps the most lengthy in the final design process, and could have easily been solved by using a simple place-and-route tool, or a gate-level simulator that could incorporate timing estimates relatively easily.

At the interfaces, a tool which could check that timing conditions (like those for the S-Pipe in Chapter 4) were obeyed and determine any available margin would have been invaluable, as well as improving confidence in the design.

4.3 Chip Signalling

The DCache communicates with the MU across almost the entire height of the die. This brings into play significant RC delay effects in the signalling between the two subsystems, and made matching the event and datapath delays difficult (indeed, a different solution was adopted that allowed additional margin to be externally allocated). The issue is complicated by the return path — event (2ϕ) signalling asynchronous systems need a return (acknowledge) event, which requires the control to traverse the interconnect path twice (four-phase systems would perform much worse in these situations, requiring four passes through a long interconnect per access cycle). In ECSTAC, this is not so great a problem because this path is to the MU, which already involves considerable latency in the return of data to the unit. However, had the MU incorporated a prefetch or victim buffer [Jou90], then the MU could be ready to send data back to the DCache very shortly after the request occurs, but the need for the acknowledge to return via the same path could stall the system. The use of more aggressive CMOS technologies could exacerbate the problem because of the predominance of interconnect delay on system speed [Max95]. Some way of mitigating this delay overhead is required.

4.4 Arbitration and Synchronisation

The design of the ECSTAC memory unit showed that the arbitration problem is expensive, and reducing the overhead may involve accepting a greater probability of failure. The overhead was partly due to the arbiter circuit used — another arbiter [AML94b] was designed using comparators, and is similar to Brunvands Q-element [Bru91], but the arbiter used avoids analog circuit design issues. Morton [Mor97, Chapter 4] also developed an arbiter based on the deferred comparison of a sampled signal, which would be interesting to evaluate in this application.

The architectural choice of using split caches was thus vindicated. Initial work on control was based on a unified cache [AM93], but the latency expense of arbitration confirmed our suspicion that shifting the contention point to a lower bandwidth interface (that is also more tolerant of latency) was a better solution.

4.5 Pipeline Control

The S-Pipe mechanism is a low overhead method of controlling asynchronous pipelines, as demonstrated in Chapter 4. However, the overhead of return event signalling and resetting the input latch line still results in an overhead of about 2 to 3 gate delays per cycle. If performance is going to be maximised, then this penalty must be mitigated to some extent, particularly as the depth of logic between pipe stages in current machines is 10 to 20 gates [Bou96, Bak90, LCT+95] in total!

4.6 Asynchronous Design Issues

The design, especially at the circuit level, showed a number of promising aspects, in addition to some undesirable qualities, when examined subsequent to its completion.

The Good Things

Perhaps the most important factor which contributed to the ability of two designers to complete such a project in the time taken [App96] was *locality*. The complete absence of any global timing constraints or well-controlled global signals meant that all units were totally self-contained and highly resilient to any slight timing variations in connected units. As long as all the system units meet predefined interface timing constraints, in this case the bundled-data constraint, the system as a whole operated correctly.

The well-considered use of self-timed blocks, like the SRAM and tag unit, can lead to very good performance if care is taken to keep control off the critical path. In blocks like SRAMs, a large amount of data can be produced at the same time, and self-timed control can switch this data out very quickly (see Table 5.2). This ability to exploit locality and bandwidth in individual units is quite attractive.

In order to exploit these factors in future implementations, localised signalling will have to be much more aggressive (in terms of timing), and faster circuits used (as opposed to many of the datapath components of ECSTAC which use static implementations), which can be made amenable to locality properties of self-timing [LCT+95].

The Not So Good Things

One of the drawbacks of asynchronous design, at least from a performance viewpoint, is its forgiving nature. Since there is no really hard cycle time *target*, it becomes all too easy to put gates in the critical path without properly considering the performance impact of this logic from a system performance viewpoint. It also becomes easy to add modules and decoupling FIFOs where they *seem* to be needed should potential for varying unit latencies be deemed particularly important. This tendency needs to be kept closely in check if performance targets are to be met, since such additions can have unexpected impacts on system performance.

The property of timing resilience mentioned above does come at a price. A performance penalty is paid for the localised signalling, and any *added* timing margin directly impacts on cycle time by adding to the critical control path. Therefore, for aggressive implementations good timing models and timing verification will become essential [YDA+97, Sch95].

Long-distance signalling is a problem, and will only become more problematic as the disparity between gate speed and interconnect delays worsens with shrinking process geometries.

Synchronous designs can loose significant fractions of their cycle time simply driving global interconnect [McL96], and in asynchronous systems this is two to four times worse due to the signalling scheme.

The road here on ...

It is interesting to apply the phrase *"in with the good, out with the bad"* after the experience of ECSTAC. A future asynchronous methodology, based on the above discussion, should attempt to retain the locality of asynchronous systems that eases design issues, and allows units to operate in an autonomous manner fully utilising self-timed circuits. However, the tendency for asynchronous control to get on the critical path should be eliminated, and some kind of timing target be used to force performance properties upon the implementation. In addition, some attention to long or system-wide interconnect lengths and their attendant delays is required if the performance of asynchronous implementations is not to be lost on communication delays in advanced CMOS technologies.

These goals are considered a *manifesto* for the succeeding chapter.

5 Conclusion

This chapter has presented the architecture and test results, in addition to the design of the cache and external interface components, of the asynchronous microprocessor ECSTAC. The design of ECSTAC involved the decomposition of the design from architectural description to circuits and complete layout, and proved to be a valuable exercise in establishing ECS as a viable methodology and identifying a path for subsequent ideas on asynchronous logic. The design of the ICache and DCache control showed that two-phase control design is not particularly problematic, and methods for improving control performance have been identified. ECSTAC was central in identifying new ideas and concepts for exploration.

ECSTAC has been fabricated and tested. Tester hardware required considerable attention to noise issues, and power and ground bounce effects caused by the speed of the signal I/Os had to be minimised. This has showed that, despite an unidentified problem with the ICache, the design is functional based on the observations made to date.

The architecture and implementation in many respects was not optimal, especially since this was a first-run project using a totally new asynchronous design approach. However, the features of ECSTAC which beg modification have been identified, primarily at higher levels of the design process. In particular, the need for proper architectural exploration cannot be stressed too heavily. A good architectural exploration, coupled with initial floorplanning

estimates and preliminary layouts, would have enhanced the performance and capabilities of the device far beyond what was actually achieved. However, in doing this the two designers involved in the project *might never have finished the design!*

Chapter 6

Free-Flow Asynchronous Systems

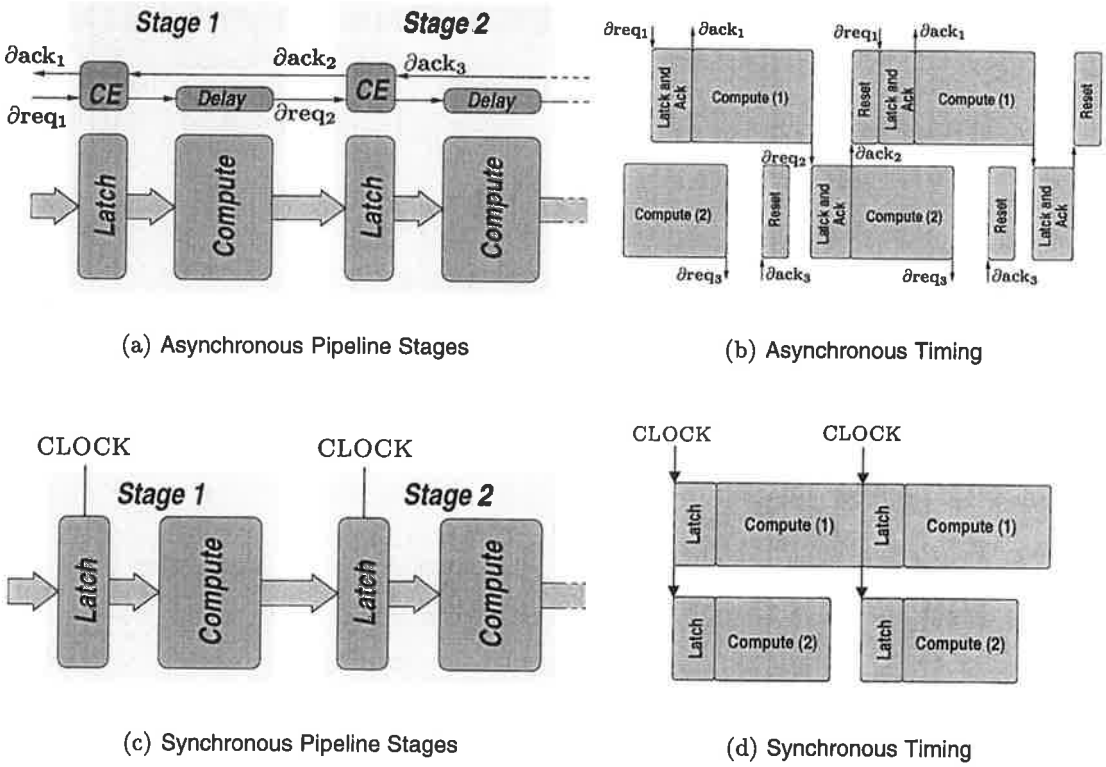
Speed isn't everything, it's the only thing.
Seymour Cray.

ASYNCHRONOUS pipelines, although conceptually elegant, suffer performance degradation due to control being on the critical timing path. Even in ECS pipeline controllers, in which a conscious effort is made to keep control off the critical path wherever possible, the cycle time achievable lags that achievable by a purely synchronous implementation (commercial devices implemented in $0.8\mu m$ technologies have cycle times down to $10ns$ [Bur], compared to $3.9ns$ for the ECS S-Pipe with no processing in Chapter 4). To offer a realistic alternative, asynchronous approaches and implementations should close this performance gap.

In this chapter, the *free-flow* technique that improves asynchronous pipeline throughput by changing the inter-stage communication mechanism is described. The circuit and system architectures necessary to maintain the performance advantage provided by the technique in a practical sense are detailed. The design of accurate and consistent delay elements is considered, as the characteristics of these elements can impact on the cycle time achievable using free-flow. Finally, the free-flow method is contrasted with existing methods for synchronous and asynchronous design.

1 Free-Flow Concepts

The design of ECSTAC revealed that the locality of asynchronous methods proved very useful in designing a large system, but that the overhead introduced by the local control tended to detract from the maximum achievable performance. To analyse the performance tradeoffs in pipelines, consider the two methods of inter-stage synchronisation shown in Figure 6.1.



(a) Asynchronous Pipeline Stages

(b) Asynchronous Timing

(c) Synchronous Pipeline Stages

(d) Synchronous Timing

Figure 6.1 Pipelines.

The locality that proved useful in asynchronous designs comes from the purely localised nature of the *req* and *ack* signals, shown in Figure 6.1(a). Since the *req* and *ack* signals uniquely determine the state of each stage, there is no need for any global signals and all control is done locally. However, there is an overhead involved (Figure 6.1(b)), since each stage must wait for the subsequent stage to latch data before resetting and accepting new data.

The performance of synchronous pipelines, like that of Figure 6.1(c), is due to the low overhead in passing data from one stage to the next. In synchronous pipelines the passing of

data from stage i to stage $i+1$ is overlapped with the passing of data from stage $i-1$ to stage i, and the handing out of data is concurrent with the receipt of new data, as shown in Figure 6.1(d). This results in almost no communication overhead and a high utilisation of logic, as the logic is only *inactive* during the latching intervals. However, in order to ensure correct operation the clock must arrive at all latching points at approximately the same time. Any difference is known as *clock skew* and causes cycle time degradation [Fri95]. This can be contrasted with asynchronous pipeline communication, where interfaces enforce sequentiality between data hand-out and data hand-in, since the sequence must be ordered

$$\partial req\,out \rightarrow wait\ for\ (\partial ack\,out) \rightarrow wait\ for\ (\partial req\,in) \rightarrow receive\ data \rightarrow \partial ack\,in$$

The issue then becomes, how can asynchronous communication be moved closer to the synchronous model in terms of signalling overheads? This will undoubtedly improve the speed and utilisation of the pipeline. Our first approach was to minimise the overhead introduced by the sequentiality of communication by moving to a BD model, resulting in the S-Pipe (and others) of Chapter 4. However, even this approach has its limits in that a certain amount of control is still required to enforce sequentiality. To go further, sequentiality is eliminated entirely, resulting in the pipeline of Figure 6.2, the basic *Free-Flow* pipeline [AML96, AML97a].

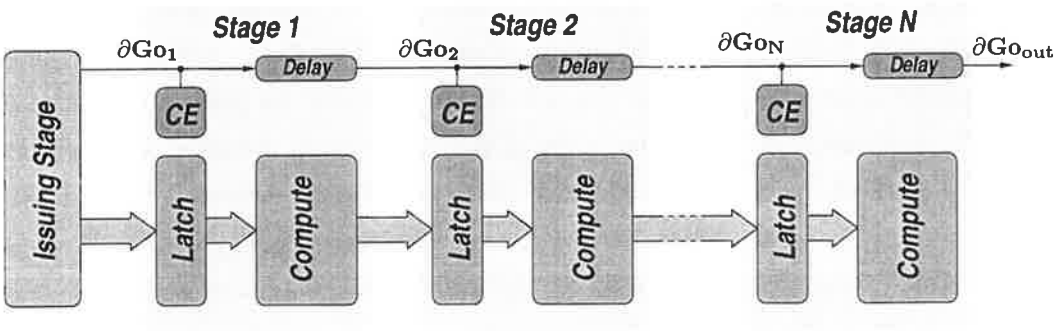


Figure 6.2 Basic Free-Flow Pipeline.

Sequentiality in control flow has been eliminated by removing the back-propagating *ack* signal of Figure 6.1(a). This allows the stage to send out data at the same time as it receives new data, thus breaking the communication ordering required in asynchronous pipelines. Although this seems unusual, in pipelines like that shown in Figure 6.1(a) or the micropipeline [Sut89], where the datapath is modelled by delays only, both the throughput and latency of the pipeline are totally static. This makes the acknowledge in such a structure redundant, because the speed at which the pipeline operates is fixed, and thus the *acknowledge* signal is irrelevant and only adds to the critical path of the circuit. To delineate between asynchronous

and *FeFA* (Free-Flow Asynchronous) pipelines, the input event signals are named ∂Go as opposed to ∂req or $\partial start$.

The change in signalling mechanism has a number of implications on the control of the pipeline. The control is now no longer part of the critical path, as the reverse propagating ∂ack, which limits cycle time by adding to the critical forward datapath latency of the datapath, is no longer present. The pipeline stage does not assert its own minimum cycle time upon its interfaces any longer, so this must now be ensured by control. The control that ensures this *requirement* now resides in the first stage of the pipeline which issues operands into the pipeline and generates the first ∂Go event — this stage is called the *Issuing Stage* (ISS). The ISS ensures pipeline constraints are met, and its design will be discussed in Section 2. The Issuing Stage also acts to set the *target* speed of the pipeline in design tasks, since the ISS will not tolerate a stage which operates slower than the issuing rate.

The control continues to use 2ϕ naming, conventions and control. This is because the free-flow method continues to use ECS techniques for design. ECS control techniques in pipelines may still be and are employed. However, using free-flow pipelines requires a fundamental shift in design methods and system architecture.

If four-phase free-flow pipelines were desired, a pulse would be needed to control the free-flow pipeline so that feed-though problems in the latches could be avoided. The propagation of a pulse through a reasonably long delay chain is not particularly reliable, as shown in Figure 6.3. The width of the pulse involves a double-sided constraint, since the pulse width must ensure adequate latching time, but must also be small enough to avoid data feed-through between adjacent latches. Pulse propagation becomes even less reliable when process variations are considered, because the pulse can *spread* either way (becoming wider or thinner depending on the process skew characteristics). Therefore, the use of pulses would require a delay element that does not distort the pulse greatly, or a pulse regenerator at every stage to reshape the pulse, which would also handle pulse *spread* though the delays. However, using edge-propagation (2ϕ) through delay elements avoids these problems to a very large extent (the impact of skewing effects is considered in Section 4.2).

1.1 Relation to existing ECS techniques

ECS promotes the *skipping* of acknowledges when latency factors allow the *ack* from one stage to be skipped [Mor97, Chapter 4.2]. An example of such a skip is shown in Figure 6.4.

In this scheme, as the internal latencies of the units are known, the acknowledge ∂ack_2 can be *skipped* safely (which otherwise would have to go to a *last* gate to produce $\partial ack_M \leftarrow \partial ack_1 \cdot \partial ack_2$) as the acknowledge of Unit-1 provides adequate timed completion for both

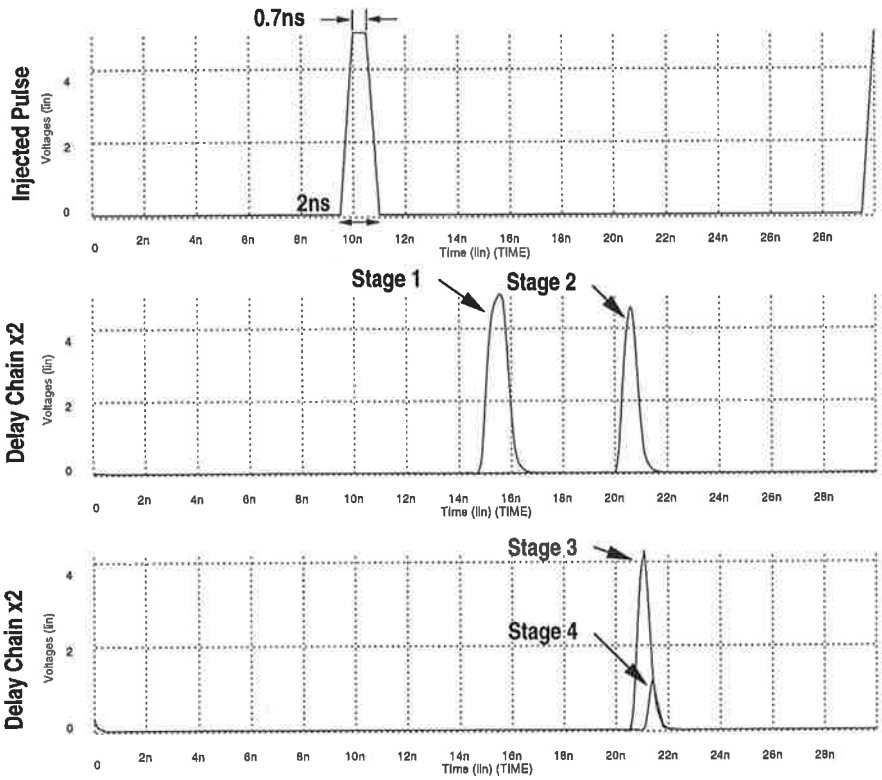


Figure 6.3 Pulse Propagation in a Delay Chain. The inverters making up the delay chain were carefully optimised for equal rise and fall delay (W_n=2μm and W_p=4.5μm), but the pulse still fails to propagate correctly all the way through the delay chain. Using a longer pulse may be more appropriate, but this shows that the pulse width can shrink to a point where the delay chain fails. Any shrink in the pulse may cause latches to fail due to insufficient latching time.

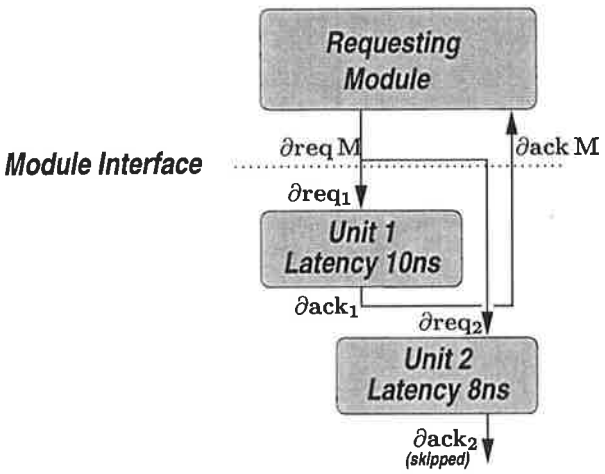


Figure 6.4 ECS Acknowledge Skipping.

units. However, in free-flow systems, the acknowledge of *neither* units is required, and instead the *Requesting Module* would be expected to behave such that the requests to the two units did not proceed too rapidly – free-flow eliminates the acknowledge explicitly.

1.2 Relation to Synchronous techniques

Some similarities may be drawn between free-flow and synchronous techniques, as both have the capability of simultaneously receiving and transmitting data from a module or pipeline stage. In fact, free-flow pipelines are equivalent to synchronous pipelines if all the delays of Figure 6.2 are identical, and the operand insertion rate is constant and equal to this delay. This is a highly specific and restrictive case, unlikely to occur in practice. The pipeline is asynchronous, and thus the operand initiation rate may vary widely. Stage latencies may vary, even dynamically (in Section 3.1). The pipeline can be stopped without loss of operands or state (in Section 2.3), which could not be done if the structure were source-clocked synchronously. Free-Flow pipelines can be interfaced easily to asynchronous systems and vice versa (in Section 3.2) if it is expedient to do so.

Thus, although free-flow may seem akin to synchronous techniques, it is more accurately a hybrid between synchronous and ECS asynchronous techniques, having the low timing overhead of synchronous implementations, and the locality properties of asynchronous systems.

2 Pipeline Design

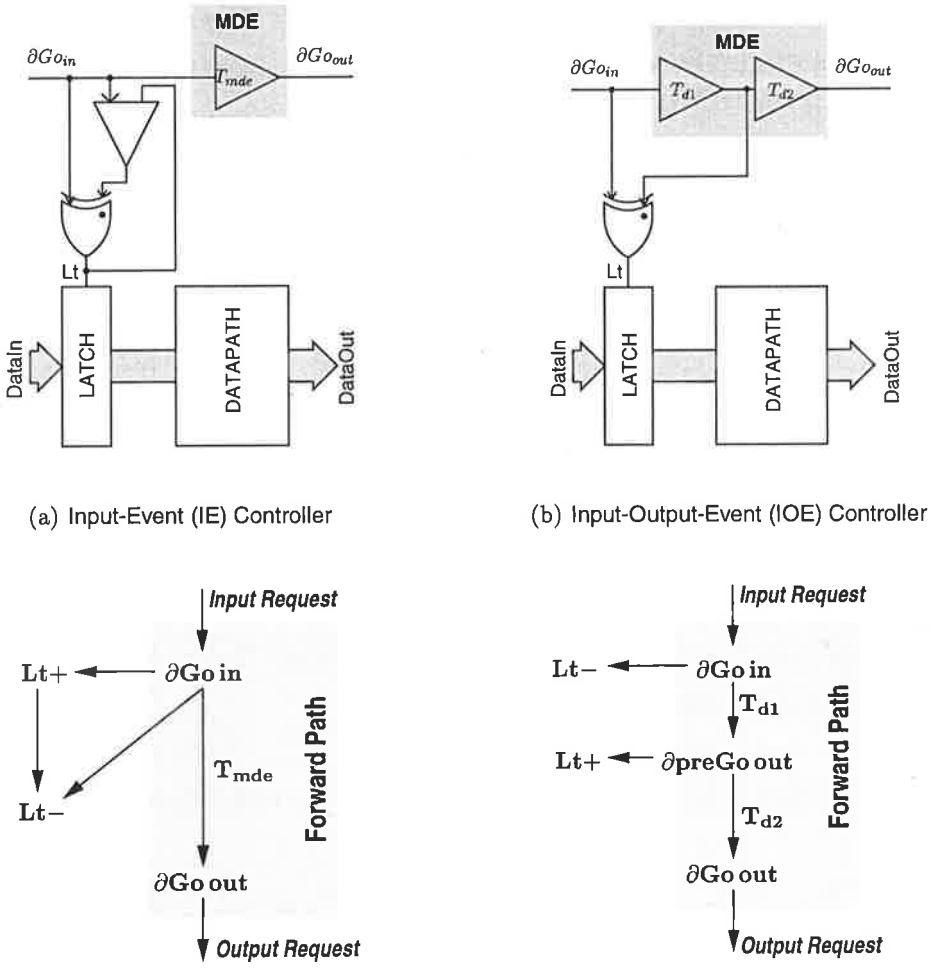
Now that the rationale behind the free-flow approach is evident, the design of the pipeline elements can be described. The delay elements used in Figure 6.2 are exactly equivalent in implementation to two-phase delays required by the ECS methodology to match module latencies. However, the elimination of sequential control dependencies requires tighter timing constraints on the operation of the pipeline stages as part of the larger system. The components of the free-flow pipeline are the *stages*, requiring control using 2ϕ signals, and the *Issuing Stage* (ISS), both shown in Figure 6.2. The design and timing constraints on both of these components is considered, before moving on to design techniques which improve the usability of the free-flow approach in general applications.

2.1 Stage Control

Each stage receives one ∂Go event at the input interface, and produces one ∂Go event at the output interface. At the individual pipeline stage level, the control must obey

$$T_{delay} \geq T_{pl} + T_{datapath}$$

The stage delay element, T_{delay}, which is the forward control latency in this case, must exceed that of the combined latch (T_{pl}) and datapath ($T_{datapath}$) components. This requirement is generally applicable as a safe constraint for both asynchronous and synchronous pipelines. The latch control signals are generated from the 2ϕ ∂Go signals. Two methods of generating the required control signals are shown in Figure 6.5. Both controllers have timing constraints that must be obeyed in order that the system operates correctly.



(a) Input-Event (IE) Controller

(b) Input-Output-Event (IOE) Controller

(c) IE simplified EFG

(d) IOE simplified EFG

Figure 6.5 Free-Flow Pipeline Stage Controllers. The Input-Event (IE) controller uses a pulse generator in the input side to control the input latch, while the Input-Output-Event (IOE) controller relies on input-output event timing to close and open the latch. Note that the IE control keeps the latch *opaque* when the stage is free, whereas the IOE control puts the latch into the *transparent* state when the stage is free.

2.1.1 Input-Event(IE) Controller

The scheme employed by the input-event (IE) controller of Figure 6.5(a) is to generate the latch control only from the input event, with the controller generating a pulse on Lt when a ∂Go_{in} event occurs. The only element in the forward control path is the *Matched Delay Element* (MDE), which nominally *matches* the forward delay of the control path to that of the datapath.

Note that if the *send* gate controlling the pulse width on Lt were placed such that it directly fed the delay element (thus $\partial \hat{Go}_{delay} \leftarrow \partial Go_{in} \cdot Lt$), then the forward delay would be dependent on the width of the latch, which is undesirable because it couples the delay element design value to specific widths of the datapath, making the design process and general timing verification harder.

Constraints

The two most basic constraints concern the maximum operating speed and the delay element value,

$$T_{MDE} \geq T_{pl} + T_{data\,i}$$
$$T_{cycle} \equiv T_{\partial Go_{in} \rightarrow \partial Go_{in}} \geq T_{MDE}$$

where T_{cycle} is the input cycle time, or the time between ∂Go_{in} events. Operands cannot be applied faster than the latency of the matched delay element, T_{MDE}(MDE). This rate constraint is the basis of the design of free-flow systems. There is also the possiblity of *feed-through*, constrained in the i^{th} stage by

$$T_{until} + T_{\uparrow Lt_i} + T_{pl} + T_{fast\,i} + (T_{cycle} - T_{MDE\,i}) \geq 2 \cdot T_{until} + T_{send} + T_{\downarrow Lt_{i+1}} + T_{\uparrow Lt_{i+1}}$$

simplifying

$$T_{\uparrow Lt_i} + T_{fast\,i} + (T_{cycle} - T_{MDE\,i}) \geq T_{until} + T_{\downarrow Lt_{i+1}} + T_{\uparrow Lt_{i+1}}$$

where T_{cycle} is the initiation interval, and thus $T_{cycle} - T_{MDE\,i}$ represents *dead-time* in the stage, which should be minimised. In the most restrictive case,

$$T_{fast\,i} \geq T_{until} + T_{\downarrow Lt_{i+1}}$$

and thus there is a required logic depth of ≈ 1.5 gate delays in the most aggressive case (no dead-time and assuming approximately equivalent latch line driver timings). This is because the method used generates a pulse on the Lt line of width $T_{send} + T_{until} + T_{\downarrow Lt}$ (significantly greater than T_{pl}), therefore latches in adjacent balanced stages are open at the same time for this pulse width, making feed-through constraints tighter.

Control-Data Timing

The *bind* operator (Chapter 4) can be used to specify the relationship between event signals and data values in free-flow pipelines. There are constraints on the input binding such that the data will still be latched, where

$$\begin{aligned} \partial Go_{in} \overset{|c}{\circ} DataIn \;&\Rightarrow \\ c \;&=\; 2 \cdot T_{until} + T_{send} + T_{\uparrow Lt_i} - T_{setup} \end{aligned}$$

where T_{setup} is the latch setup time. The output binding of the pipe stage is

$$\begin{aligned} \partial Go_{out} \overset{v}{\circ} DataOut \;&\Rightarrow \\ v \;&=\; (T_{pl} + T_{data\,i} - T_{MDE\,i}) + min(v_{min}, \mathcal{B}(\partial Go_{in} \circ DataIn)) \\ \text{where} & \\ v_{min} \;&=\; T_{until} + T_{\uparrow Lt_i} \end{aligned}$$

Thus, the IE controller will always generate some finite control-data skew unless the T_{mde} element is overdesigned (with $T_{mde} > T_{data\,i} + T_{pl}$) to bring the skew back towards the bundled-data constraint, however this will result in a non-optimal throughput. Conversely, the matched delay element (MDE) can be slightly underdesigned in limited parts of the design if it can be allowed from the binding constraints.

IE Pulse Control

The pulse control for the IE controller presently uses a circuit that generates a pulse width of at least T_{pl} because its feedback path contains a latch identical to that found in the datapath (the *send* gate). However, this creates a minimum logic depth requirement in the general case to avoid feedthrough. An alternative method could be used if this minimum logic depth requirement will prove problematic. A *delay* gate used to reset the *until* gate, instead of feeding back the input event via a *send* gate, will create a narrower pulse at the expense of designing a new delay and verifying that it creates an adequate pulse-width considering parameter variations. Alternatively, the *until* gate output, which typically drives a buffer for the heavily-loaded line *Lt*, could be fed directly to the *send*. This bypasses the dependence on the time taken to drive the latch control line from the control path, decreasing the duration of the latching pulse, and thus decreasing the minimum logic depth requirement.

2.1.2 Input-Output-Event(IOE) Controller

The control schema of Figure 6.2 shows that two event signals are used in each stage, one to start the input (∂Go_{in}), and one to start the output (∂Go_{out}). These two signals can be used directly to control the stage latch, opening the latch when the stage is inactive. This control structure is shown in Figure 6.5(b), the input-output-event (IOE) controller.

Constraints

The primary constraints on the delays in the stage are,

$$\begin{aligned} T_{cycle} \equiv T_{\partial Go_{in} \rightarrow \partial Go_{in}} &\geq T_{d1} + T_{d2} \\ T_{d1} + T_{d2} &\geq T_{pl} + T_{data\,i} \end{aligned}$$

which is identical to the IE controller, with the two delays lumped together. The feed-through constraint is

$$\begin{aligned} T_{d1} + T_{until} + T_{\uparrow Lt_i} + T_{pl} + T_{fast\,i} &\geq T_{d1} + T_{d2} + T_{until} + T_{\downarrow Lt_{i+1}} \\ \text{simplifying} & \\ T_{pl} + T_{fast\,i} &\geq T_{d2} \end{aligned}$$

therefore, if $T_{d2} \approx T_{pl}$ (see below), the fastest logic path can be approximately *zero*. Increasing the value of T_{d2} from this value worsens the fast-path constraint by requiring $T_{fast\,i} > 0$, from the above constraint equation. A further constraint exists (not necessary in the IE controller) on the *response* of the IOE controller, that affects the values of T_{d1} and T_{d2}. The latch line Lt_i must be high for long enough to latch input data before a new ∂Go_{in} arrives,

$$\begin{aligned} (T_{d1} - T_{cycle}) + T_{until} + T_{\uparrow Lt_i} &\leq T_{until} + T_{\downarrow Lt_i} - T_{setup} \\ \text{if } T_{cycle} = T_{d1} + T_{d2} & \quad \text{then } \Rightarrow \\ T_{d2} &\geq T_{setup} \\ &\gtrsim T_{pl} \end{aligned}$$

To ensure the input latch resets in a timely manner, a minimum value of T_{d2} is required of approximately T_{pl}, and using this value also gives the value of $T_{fast\,i} \geq 0$ that avoids any feed-through problems (however, note that the constraint on T_{d2} is double-sided, and widening latch pulse width requires greater attention to be paid to feed-through issues). The required value of T_{d1} is then $T_{data\,i}$, the datapath propagation delay.

Control-Data Timing

The constraint on the input ∂Go_{in} to $DataIn$ timing is very tight,

$$\begin{aligned} \partial Go_{in} \overset{|c}{\circ} DataIn &\Rightarrow \\ c &= T_{until} + T_{\downarrow Lt_i} - T_{setup} \end{aligned}$$

In practice, this value of input skew would best be approximated to zero — the IOE controller *requires* the bundled-data constraint for correct operation. The output binding is simple,

$$\begin{aligned} \partial Go_{out} \overset{v}{\circ} DataIn &\Rightarrow \\ v &= (T_{pl} + T_{data\,i} - T_{mde\,i}) + \mathcal{B}(\partial Go_{in} \circ DataIn) \end{aligned}$$

There is very little margin for underdesigning the value of the MDE because of the tight constraint on the input event-data timing. Therefore, to successfully interface an IE to an IOE controller will require either a dummy stage to re-align control and data to the bundled-data constraint, or some added time in the IE stages' MDE element, potentially impacting on system cycle time. The converse interface, IOE to IE, has no timing issue because the IOE controller maintaints the bundled-data constraint at its output.

2.1.3 Controller Design Assumptions

To simplify the MDE design values, it was assumed that a property of the implementation is the relatively good matching in drive times of the latch lines, and these considerations can then be factored out. However, if this is not the case or more detailed constraints are required, then these differences will have to be included. The design of each MDE will change such that

$$T_{MDE_i} = T_{pl} + T_{datapath\,i} + (T_{\uparrow Lt_i} - T_{\uparrow Lt_{i+1}})$$

which properly expresses the forward datapath latency. The binding constraints would also need to be modified to include this effect.

2.1.4 Dynamic Logic in the Datapath

The two free-flow controllers can be adapted to use dynamic logic in the datapath, for which an *evaluation* signal is required, as shown in Figure 6.6. The constraints for these two controllers are developed similarly to those for the static logic case. Both assume that the ISS slows its issuing rate to allow precharging to complete before the next operand arrives.

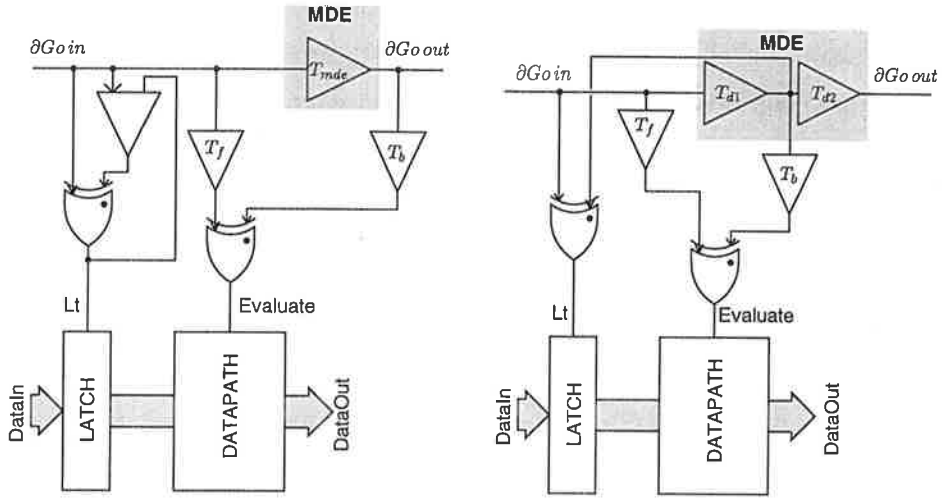
The Input-Event controller can achieve a cycle time of

$$T_{cycle} = T_{evaluate\,i} + T_{precharge} + T_{setup} + T_{hold}$$

where T_{setup} represents the time data must be valid before a $\downarrow Lt$ and still be latched (typically $\simeq T_{pl}$), and T_{hold} represents the time after $\downarrow Lt$ that data must be held (typically zero).

The delays necessary to obtain this performance have complex values. This is because the latch in the IE controller is opaque when the ∂Go_{in} event arrives, and thus T_f must ensure the evaluation does not commence until data is *guaranteed* to be valid,

$$\begin{aligned} T_f &= T_{send} + T_{\uparrow Lt_i} + T_{\downarrow Lt_i} + T_{until} - T_{setup} + T_{pl} - T_{\uparrow eval} \\ T_b &= T_{until} + T_{send} + T_{\uparrow Lt_{i+1}} + T_{\downarrow Lt_{i+1}} + T_{hold} - T_{\downarrow eval} \\ T_{MDE\,i} &= T_{pl} + T_{eval\,i} \end{aligned}$$



(a) IE dynamic controller

(b) IOE dynamic controller

Figure 6.6 Dynamic Logic Controllers. These are modified versions of the original controllers in Figure 6.5.

The delay T_b is similarly complex as it must assumes the worst-case input event-data timing. However, once the exact event-data timing in an implementation using this controller is known, the delay design can be eased by using specific knowledge from the design. For Input-Output-Event controller, the cycle time is similarly

$$T_{cycle} = T_{evaluate\,i} + T_{precharge} + T_{setup} + T_{hold}$$

However, the delay design is considerably easier in this case as the IOE controller demands bundled-data timing on the input (and output) sides of the stage,

$$\begin{aligned} T_f &= T_{until} + T_{\uparrow eval} - T_{pl} \\ T_b &= T_{\downarrow Lt_{i+1}} + T_{hold} - T_{\downarrow eval} + T_{d2} \\ T_{d1} + T_{d2} &= T_{pl} + T_{eval} \end{aligned}$$

which are much easier to design than the delays for the IE controller. In addition, because the stage is guaranteed to be free for a set time after evaluation (as the ISS issues at a rate that allows all stages to precharge before the next request), the design of T_{d2} is considerably eased, and can set to zero in most implementations.

Both controllers achieve a performance equivalent to the optimal ECS dynamic controller, $PP\alpha$ [Mor97]. This is because ECS can hide the reset phase overhead incurred by the asynchronous paradigm in the time taken to precharge the stage logic when using totally bounded-delay pipelining.

2.2 Issuing Stage

The Issuing Stage (ISS) issues operands into a pipeline composed of free-flow stages. The design of the free-flow stage controls always required that

$$T_{cycle} \equiv T_{\partial Go_{in} \rightarrow \partial Go_{in}} \geq T_{MDE\,i}$$

the rate at which operands arrive at the stage inputs is no greater than the value of the matched delay. Therefore, the ISS, which repeatedly generates the event ∂Go_1, should ensure in an N-stage pipeline that

$$T_{\partial Go_1 \rightarrow \partial Go_1} \equiv T_{cycle} \geq max(T_{MDE\,1}, T_{MDE\,2}, \ldots, T_{MDE\,N})$$

Operands are issued into the pipeline *no faster* than the slowest stages' cycle time. Any conventionally pipelined system, be it synchronous or asynchronous, is limited to this value in the steady state. This interval, $T_{\partial Go_1 \rightarrow \partial Go_1}$, is termed *issuing rate*, T_{issue}, and is the operand initiation rate into the free-flow pipeline. The *Issuing Stage* (ISS) is usually integrated with the first functional stage of the pipeline. For example, the ISS in a microprocessor would be the Program Counter before the Instruction Fetch stage. Although the functionality embodied by the ISS at present seems trivial, in later expansions of the free-flow methodology, the ISS becomes a critical system component in determining system throughput.

2.2.1 Issuing below the Peak Rate

As every stage sees the same rate of operand arrival, slowing the issuing rate below the peak achievable creates *dead-time* in pipe stages,

$$T_{dead\,time_i} = T_{issue} - T_{MDE_i}$$

In the general case, dead-time should be avoided as it slows throughput. However, if stages have a particular requirement for dead-time to precharge (for example, dynamic logic stages), then the ISS can be used to factor in this precharge time on a global level by slowing the issue rate.

2.3 Halt Logic

It may immediately be noted that the pipeline of Figure 6.2 is *inherently* unstoppable, and a stop-condition in the i^{th} stage will result in the total failure of earlier stages of the pipeline that are *expecting* to send their operands forward into the pipeline. Two methods of avoiding such occurrences are the *broadcast* halt mechanism and the *propagated* halt mechanism.

2.3.1 Broadcast Halt

One method of generating the pipeline halt signal is simply to *broadcast* it, globally, when a long-latency condition is detected, shown in Figure 6.7.

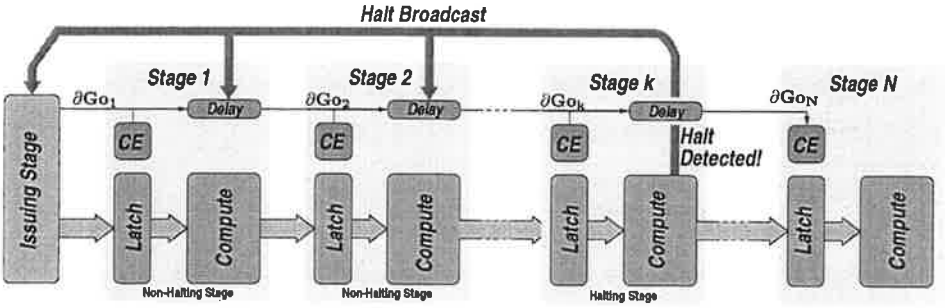


Figure 6.7 Broadcast Halt Scheme.

A stage which may halt (due to a long latency operation or an error condition, for example) asserts a global halt signal, broadcasting it to every previous stage causing them to *stop*. A modification to the IE controller to support this functionality is shown in Figure 6.8.

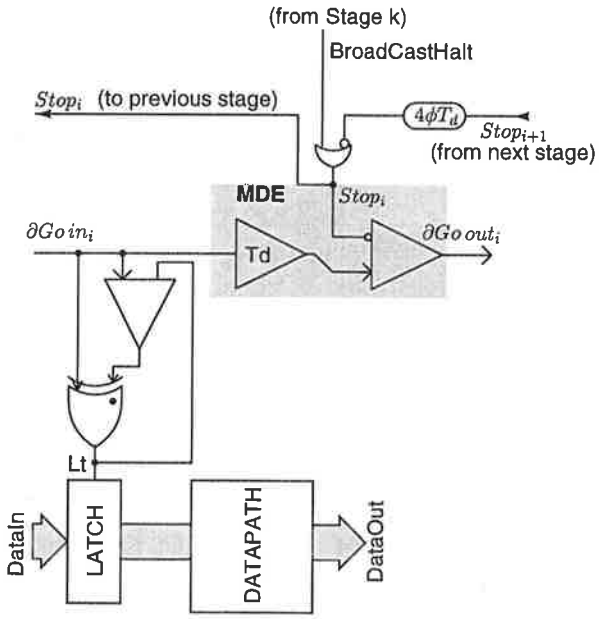


Figure 6.8 IE controller with Broadcast Halt functionality.

If the k^{th} stage is asserting the halt, then only stages $k-1$ to the ISS receive the broadcast halt signal so generated. The stage asserting the halt condition generates $\uparrow BroadCastHalt$, which takes every $Stop_i$ signal low ($i \in 1 \ldots k-1$), through the NOR gate of Figure 6.8, in each stage. This halts the issuing of any further ∂Go signals in the pipeline. Note that the potential for metastability failure exists because the assertion of the $BroadCastHalt$ signal is uncorrelated to activity in the individual stages.

The delay element between the de-assertion of the halt condition (indicated by $\downarrow BroadCastHalt$ in the circuit) and the re-start of the pipeline control ($\uparrow Stop_i$) is so that the pipeline does not fail during halt exit. When the halt is asserted, every stage must be assumed to be processing an operand at the time (this is the most restrictive case), and when the halt *exits*, every stage will have a ready, valid operand to pass to the next. If this were allowed to happen as soon as $\downarrow BroadCastHalt$ occurred, the pipeline could potentially fail by the following mechanism. Assume there are two adjacent stages with MDE values $T_{MDE\,j}$ and $T_{MDE\,j+1}$, then if the pipeline was allowed to exit by simply pulling up every *StopOut* signal, then every stage would issue an operand to the succeeding stage at the same time. If $T_{MDE\,j} < T_{MDE\,j+1}$ then stage $j+1$ will see an operand at $t = 0$ (when the halt exits), and then another operand arrives at $t = T_{MDE\,j}$, which breaks the issuing rate constraint at this stage, causing failure. Thus, these reverse-going delay elements are required and the value in the j^{th} stage is

$$T_{4\phi delay\,j} \geq max(T_{MDE\,j+1}, \ldots, T_{MDE\,N}) - (T_{MDE\,j+1} + T_{nor})$$

as the rippling effect ensures that the constraint is met *iteratively*. Thus, if stage j has a MDE close to the value of the pipeline cycle time, the delay can be omitted, although keeping $\overline{Stop_j} = BroadCastHalt + \overline{Stop_{j+1}}$ keeps the rippling effect to the $\uparrow StopOut_j$ transitions, giving extra margin to the exit of the halt condition. Alternatively, the halt signal could be delayed only locally, in which case the value of the j^{th} delay would be

$$T_{4\phi delay\,j} = \sum_{i=j+1}^{k} (T_{MDE\,k} - T_{MDE\,i})$$

where k is the worst-latency stage between the j^{th} stage and the end of the pipeline. Thus, when the $\uparrow Stop$ signal is generated locally, the size of the delay element controlling restart grows linearly with the distance from the halting point, which is quite undesirable. The rippling solution described above does not suffer from this problem.

2.3.2 Propagated Halt

The broadcast halt mechanism can halt a large number of stages at the risk of reliability failures caused by metastability in the stage control. The *propagated* halt scheme avoids any reliability failures at the expense of being able to halt only a limited number of stages. An overview of the scheme is shown in Figure 6.9.

A detected halt in Stage M ripples back up the pipeline, only passing from stage i to stage $i-1$ when the $Busy_i$ signal is high (set when the stage is busy). In this way, the halt only propagates to the previous stage when data has arrived, and the possibility of metastability is avoided. However, a constraint exists on the number of stages that may be halted, as the halt may only ripple a set timing distance until the arrival of the *Stop* signal and respective ∂Go are potentially coincident or *too late*, causing failure.

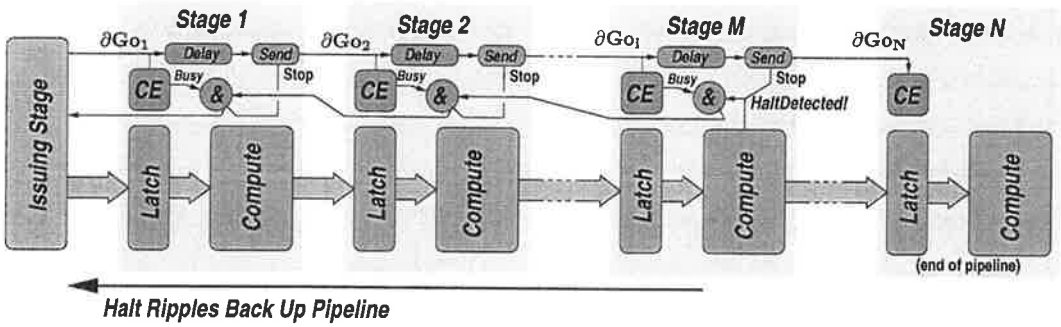


Figure 6.9 Propagated Halt Scheme. The *Send* elements are ECS *send* gates that halt the flow of control when a halt arrives, and combined with the *Delay* elements form the stage MDE value. The '&' element ANDs the *Busy* signal and the arriving *Halt* signal, propagating the halt and disabling the *send* gate when the AND condition becomes true.

Propagating Halt Constraints

The constraints can be written in an iterative fashion for the general case of multiple halt assertions in the pipeline,

$$T_{halt\,ok_i} = T_{pre\,halt_i} - T_{margin_i} - (T_{sa} + T_{hold}) \tag{6.1}$$

$$T_{margin_i} = max(T_{busy}, T_{margin_{i+1}} - T_{issue} + T_{MDE\,i} + T_{hp}, T_{hd_i}) \tag{6.2}$$

where

$$T_{margin_M} = max(T_{busy}, T_{hd}) \tag{6.3}$$

where

$T_{pre\,halt_i}$	part of MDE between ∂Go_{in} and the *send* gate halting the stage
T_{hd_i}	time to detect a halting condition in stage i, zero if not required
T_{sa}	time to assert *Stop* to *send* gate after stage busy and halt arrives
T_{hold}	*send* gate hold time, margin between $\downarrow Stop_i$ and the input event
T_{issue}	Issuing Stage operand insertion rate
T_{MDE_i}	matched delay element value in stage i
T_{busy}	time to assert *Busy* signal in each stage
T_{hp}	time to propagate halt signal from one stage to the next

The successful assertion of the propagated halt in each stage requires that $T_{halt\,ok_i} \geq 0$. The operation of the halt and the associated constraint is shown in Figure 6.10, with only the M^{th} stage asserting a halt condition. When the halt is detected, it halts stage M and propagates to the previous stage. The $M - 1^{th}$ stage is busy, so the halt propagates again. However, in the $M - 3^{th}$ stage, the operand has not yet arrived at the stage because stage $M - 3$ has a

low latency, therefore, the propagating halt waits for the stage *busy* signal to be generated before propagating further.

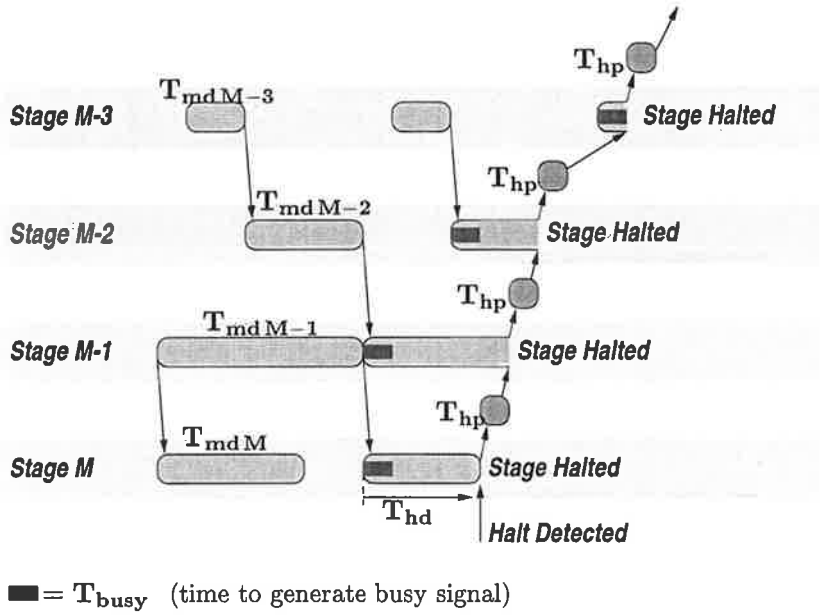


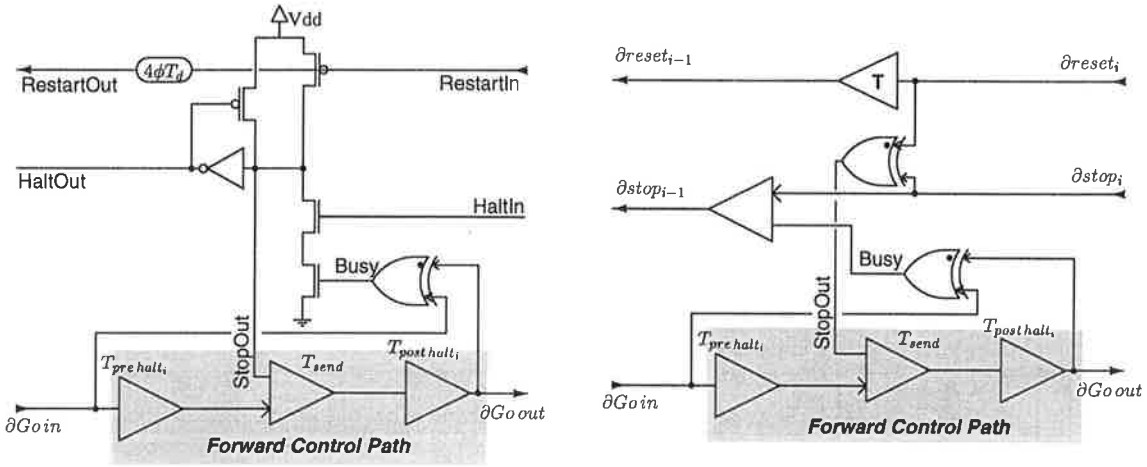
Figure 6.10 Halt Propagation Timing.

For example, in a balanced pipeline with a 15 gate delay cycle time, if the M^{th} stage takes 5 gate delays to *detect* the halt then the $M - 8^{th}$ stage (a 9-stage pipeline) may be halted safely (if $T_{hp} = T_{sa} + T_{hold} = 1$ gate delay).

The constraint on timing margin, $T_{margin\,i}$, is eased if there is *dead-time* in any stage. If the cycle time of the ISS is such that

$$T_{issue} - max(T_{MDE\,1}, T_{MDE\,2}, \ldots, T_{MDE\,N}) \geq T_{hp}$$

in which there is timing margin embedded in the issuing rate (either for safety or to allow for precharge), then if this margin exceeds T_{hp} (the time to propagate the halt signal from one stage to the next), a pipeline of *arbitrary* depth can be halted because the constraints in Equations 6.1-6.3 are continuously satisfied. In dynamic logic pipeline (in which $T_{issue} - T_{MDE\,i} \approx T_{precharge}$ is likely to be greater than T_{hp}) the timing margin stays almost constant as the halt ripples up the pipeline, allowing an arbitrary length pipeline to be halted without potential for failure.



(a) Halt Circuit using Level-Driven (4ϕ) Reset Signals

(b) Halt Circuit using Event (2ϕ) Halt and Reset Signals

Figure 6.11 Propagating Halt Circuits. The 4ϕ circuit, (a), involves more transitions on control lines than the 2ϕ circuit in (b), but will be faster due to the speed of the dynamic gate.

Circuit Implementations & Halt Exiting

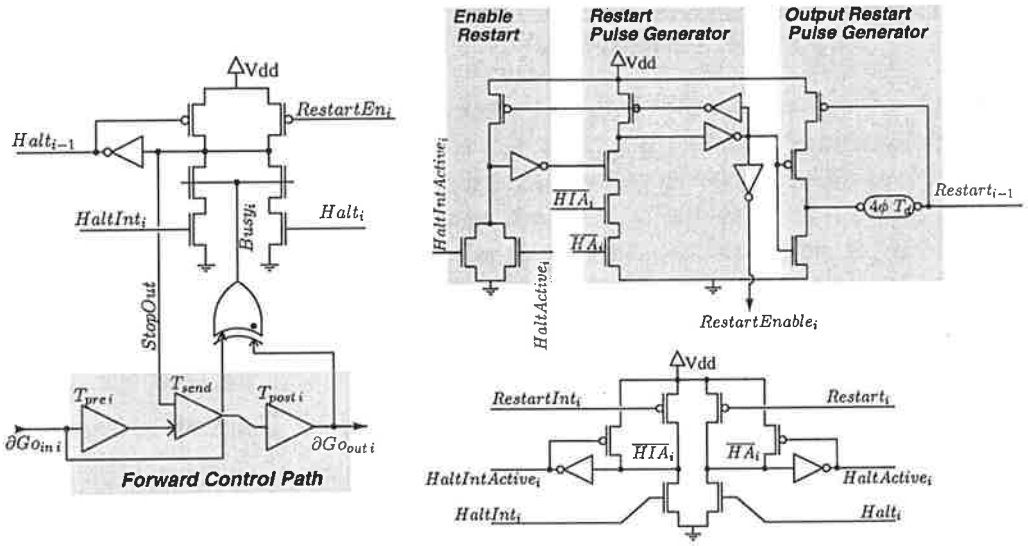
Two methods for the circuit implementation of the halt circuits are shown in Figure 6.11.

The circuit of Figure 6.11(a) propagates the halt through a dynamic logic gate. When the halt signal, $HaltIn$, arrives, and the stage becomes busy, the *send* gate is halted (via *StopOut*) and the dynamic gate activates, driving $\uparrow HaltOut$ to the previous stage. The succeeding stage is assumed to be driven in a similar manner, so that even if the first $Halt$ signal into the pipeline goes low, the dynamic gate will continue to hold the $HaltOut$ signals high. When the reset signal arrives, $RestartIn$, the dynamic gate is recharged and the $RestartOut$ signal propagated to the previous stage. The two-phase circuit of Figure 6.11(b) functions in an identical manner, using two-phase ECS circuit primitives to implement the required halt actions.

When a stage can be halted by a propagating condition *and* can assert its own halt condition, the control is more involved. A circuit implementation of the required functionality is shown in Figure 6.12. The *assertion* of either halt, Figure 6.12(a), is relatively simple. The propagating halt condition, $Halt_i$, and the internal halt condition, $HaltInt_i$, can both trigger the *StopOut* signal, halting this stage and beginning halt propagation. *Restarting* the pipeline, Figure 6.12(b), is much more complex. A circuit must be used to *remember* which halt conditions have been asserted (one or both may be asserted), and the *Enable restart*

circuit only enables the *Restart Pulse Generator* when the appropriate restart signals have been received. This restarts the i^{th} stage, and generates a pulse on $Restart_{i-1}$ when the appropriate restart delay has elapsed.

The reset-halt delay elements that delay the restart of the $i-1^{th}$ stage after the i^{th} restarts are required to ensure that the logic exits from the halt condition without failing, similarly to the *broadcast* halt case. These 4ϕ delay elements are designed using the same method as the broadcast halt case.



(a) Halt Propagation

(b) Restart Generation

Figure 6.12 Self-Asserting Propagated Halt Control Circuit. (a), the halt propagate circuit is quite simple, but the restart circuit, shown in (b), is complex because one or both conditions may be asserted when *Restart* signals arrive. $Halt_i$ is assumed to be asserted before an active $HaltInt_i$ moves to the restart phase, thus, halt processing is expected to be long compared to the time to propagate a halt to the i^{th} stage.

2.4 Analysis and Performance

The cycle time of free-flow pipelines for the static logic case is constrained to be $T_{pl}+T_{datapath}$ in the worst-case stage. This is an improvement of at least 1.5 gate delays over the D-Pipe (Chapter 4), and an improvement of at least 3 gate delays over the S-Pipe. The relative performance of free-flow pipelines will improve as the width of the datapath increases, because these factors do not enter into the design constraints for free-flow controllers.

2.4.1 Comparisons

The performance of an IE-type free-flow controller was evaluated against the S-Pipe using circuit simulation of extracted layouts. The results are shown in Table 6.1.

Metric	IE-Free-Flow	S-Pipe
Cycle Time	8.6ns	11.2ns
Latency	24.9ns	24.3ns
Restart	4.1ns	9.2ns
Utilisation	92%	70%
Power-Time (mW·ns)	166	145

Table 6.1 Free-Flow and ECS Pipeline Performance Comparisons. The results are from HSPICE simulations of extracted 0.8μm CMOS layouts, using a Level 13 model at 75^{o}C and 5V, with nominal process parameters [PP94]. Both designs used an identical delay element (with minimal delay skew, see Section 4.2) of 6.6ns (equivalent to a logic depth of approximately 8 gate delays), and a latch with T_{pl}=1.3ns of width 64 bits, integrated in a 3-stage pipeline.

A variety of different metrics can be used to compare the performance of the two schemes. Latency figures are nearly equal in this case because of the identical forward latency of the two pipelines. However, the elimination of the ∂ack signal results in a 23% cycle time improvement in the free-flow pipeline. The lower restart time of the IE controller is due to the use of identical delays in each pipeline stage, thus not requiring reverse-propagating delays on the *restart* path (see Section 2.3), which is generally not applicable. The utilisation of the pipeline logic (measured as $(T_{pl}+T_{datapath})/T_{cycle}$) is also significantly higher for the free-flow controller (theoretically able to approach 100%) due to the absence of dead-time. Power-time figures are approximately equal, although the addition of one *send* gate to the IE controller over the gate count of the S-Pipe contributes to marginally higher power consumption.

2.4.2 Why use Free-Flow pipelining?

Free-Flow pipelines can achieve a greater operating rate than normal ECS pipelines, as the above comparison has shown. Their cycle time is invariant as the width of the latch increases, and their utilisation of enclosed logic is higher. This does come at the expense of greater requirements for timing verification and control, but this is perhaps the ultimate price of obtaining higher performance levels from asynchronous logic.

Another interesting feature of free-flow pipelines is the presence of a *target* cycle time, and a control structure which encourages minimal intervention. The design of ECSTAC showed that it was far too easy and forgiving to place extra control on the critical path when the

circumstances warranted it, and since no hard cycle time target exists, this can be deemed acceptable. However, in free-flow the pipeline must operate at the rate set by the ISS. Adding extra control is either not possible, or discouraged as it disrupts the normal flow of control.

2.5 Pipeline Composition

Building a system out of free-flow stages is not trivial in general because the asynchronous nature of the computation means that arbitrary interconnections are not possible. Synchronous systems make arbitrary connections relatively easy because different stages in a synchronous machine are known to have implicit and dependable timing relations. However, in free-flow pipelines and asynchronous pipelines, implicit timing is not available between physically separate stages, making composition of stages more difficult.

2.5.1 Pipeline Forks

The general pipeline fork topology considered here is shown in 6.13.

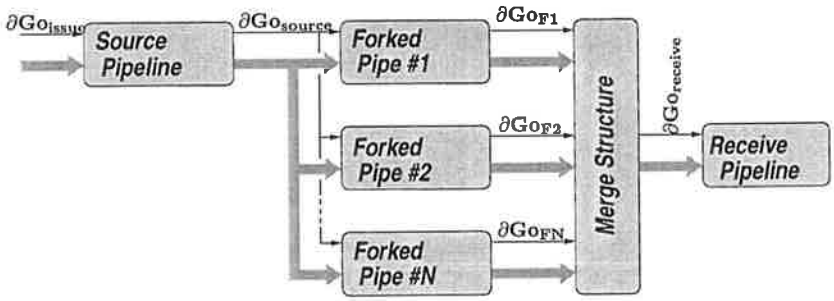


Figure 6.13 General Free-Flow Pipeline Fork.

In general, a source pipeline forks to N sub-pipelines, which then terminate into one receive pipeline. The source pipeline is usually not an issue. It must distribute data to the N forked pipelines, which will involve some wiring issues and ensuring that the binding constraints on the forked pipelines are satisfied. However, the structure which *merges* the data at the end of the forked pipelines is a challenge. The forked pipes may be unknown in exact total latency, and they may have *varying* latency (discussed in Section 3.1), which the *merge* structure must cope with.

Only forks of width two are considered here. This serves to illustrate the general mechanism of the *merge* design, which can then be readily extended to wider fork widths when required.

Fixed Latency Asymmetric Case

The general mechanism for a fixed latency forked pipe merging structure is shown in Figure 6.14. In the asymmetric case, the latency of one of the forked pipelines dominates, and the latency of the *longer* pipeline is constant. In this case, the total latencies of the two pipelines are defined as

$$\begin{aligned} T_{tl-fp1} &= T_{\partial Go_{source} \rightarrow \partial Go_{F1}} \\ T_{tl-fp2} &= T_{\partial Go_{source} \rightarrow \partial Go_{F2}} \end{aligned}$$

T_{tl-fp1} is the total latency of pipe one, and T_{tl-fp2} is the total latency of pipe one. Without loss of generality, it is assumed that $T_{tl-fp1} > T_{tl-fp2}$. Therefore, the total number of operands that can emerge from pipe two before the first pipe one operand emerges is

$$N_{outstanding\,FP2} = \left\lceil \frac{T_{tl-fp1} - T_{tl-fp2}}{T_{issue}} \right\rceil$$

These outstanding operands must not be lost if the source pipeline stalls, which requires some method of data buffering, shown in Figure 6.14.

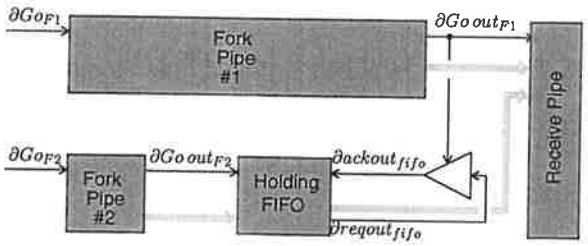


Figure 6.14 Fixed Latency Asymmetric Fork Merging.

The asynchronous FIFO (using an S-Pipe or similar structure) holds operands arriving early out of the short second pipeline until the longer first pipeline produces a matching result, which are fed to the receive pipeline and the FIFO output acknowledged. The length required of the FIFO is approximately $N_{outstanding\,FP2}$, however if the disparity between pipeline lengths is great, then the bubble propagation time in the FIFO may become significant and the FIFO lengthened to guarantee operand insertion at the pipe two interface, with the length being

$$N_{FIFO} = N_{outstanding\,FP2} + \left\lfloor \frac{T_{issue}}{T_{hp}} - (N_{outstanding\,FP2} - 1 + \frac{T_{restore}}{T_{hp}}) \right\rfloor$$

An alternative would to using a 2ϕ FIFO is a ring FIFO, whose structure is shown in Figure 6.15. This would largely mitigate any problems with bubble propagation.

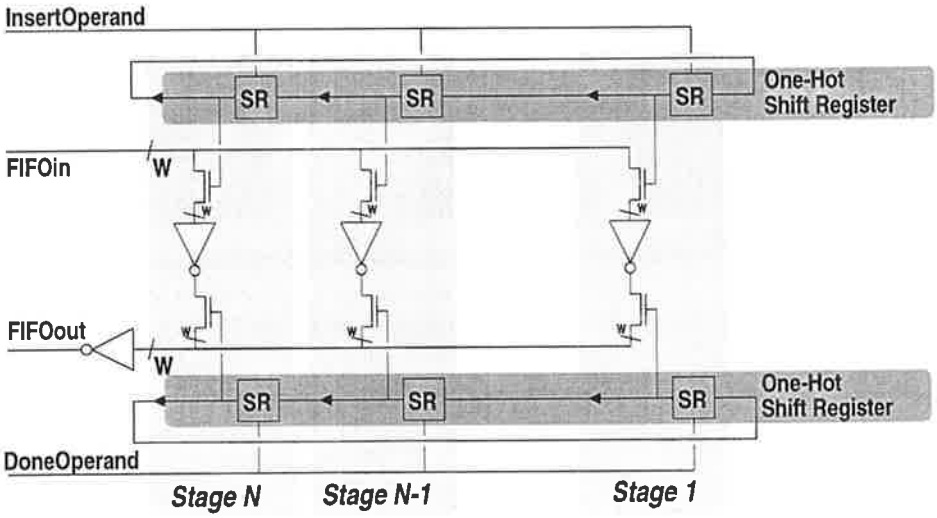


Figure 6.15 Ring FIFO. The FIFO employs two *one-hot* ring-connected shift registers that control the movement of data around the latches, which share input and output buses. The latency through this circuit when the FIFO is empty is superior to a 2ϕ S-Pipe solution, and there are no problems with bubble propagation. The circuit shown use dynamic charge storage, and could be made fully static by using weak feedback inverters across the individual stages' inverters.

Variable Latency Case

The variable latency case implies that the exact latency of both pipelines is unknown. A similar scheme using FIFOs to catch early operands cannot be used here because, if the latencies are variable, then although the *average* rate of operands emerging from the FIFOs will be T_{cycle}, the *instantaneous* rates will vary widely and will not permit proper triggering of the *receive* pipeline at an appropriate rate. One approach is to delay the input ∂Go_{source} event for the maximum of the individual forked pipeline worst-case latencies ($T_{max\,FP1}$ or $T_{max\,FP2}$), and use this to synchronise the outputs of two FIFOs constructed to hold the worst-case outstanding-operand disparities between the two pipelines. Another approach is to use a multi-rate pipeline, discussed in Section 6.

2.5.2 Data Backwarding

Data backwarding in free-flow pipelines is relatively easy because latency reductions help in the meeting of backwarding timing issues. The general scheme of data backwarding in free-flow is shown in Figure 6.16.

Backwarding is defined as a result flowing to a consuming stage. In this case, backwarding is defined as the j^{th} stage requiring the results of the k^{th} stages' result on the L^{th} operand (where $k > j$), when j is about to operate upon the $L+m^{th}$ operand. The ability to send data back up the pipeline then becomes a tradeoff between latency and topology. The constraint

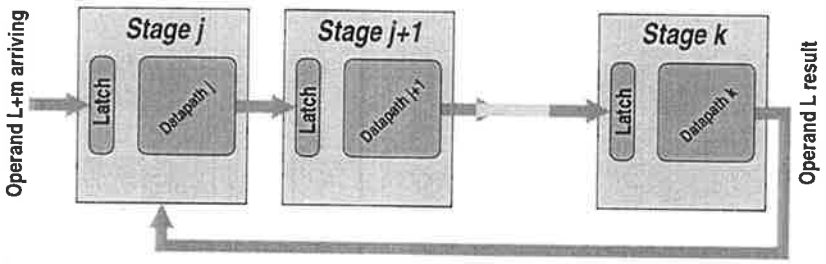


Figure 6.16 General Case Data Backwarding Scheme.

for data to be backwarded successfully becomes

$$T_{buffer} + \sum_{i=j}^{k} T_{MDEi} \leq m \cdot T_{issue}$$

The time taken to propagate from the input of the j^{th} stage to the output of the k^{th} stage, in addition to the time to pass through any added *buffers*, T_{buffer}, must be less than the time taken for the *consuming* operand to arrive. If $m = (k - j + 1)$, which, in a synchronous pipeline, would correspond to the output operand being *immediately* consumed by the j^{th} stage, then this constraint becomes

$$T_{buffer} \leq \sum_{i=j}^{k} (T_{issue} - T_{MDEi})$$

The buffer latency to get the produced operand from the k^{th} stage to the j^{th} stage must be less than the sum of the *margins* in each stage between the two points that pass data.

Buffer Sizing

A buffer is required in the backward path to absorb any operands that are not consumed by the j^{th} stage, and must be held by the buffer for later consumption. This will happen if the ISS of the pipeline slows or stalls, creating a wide bubble in the pipeline which causes a number of operands to *spill* into the buffer. If the loss of operands under such circumstances is deemed satisfactory, the buffer can be sized minimally. If not, then the buffer length (the number of operands which must be stored) is

$$N_{buffer} = m$$

Buffer Implementation and Initialisation

Using a 2ϕ FIFO for the buffer is one solution, but the design would be severely complicated because the first m operands should not attempt to remove a stored operand from the buffer, thus, a counter is required in the j^{th} stage to track operand use. In addition, the latency through a 2ϕ FIFO grows linearly with FIFO length, and thus for large values of m the buffer

latency will contribute significant overhead to the ability to meet the constraint equation for backwarding. Therefore, a ring FIFO is again used (see Figure 6.15), having a low latency when empty, which does not grow with buffer size. In addition, the initialisation issue can be handled by setting the value of the *DoneOperand* shift register to ensure that when the first *real* backward is attempted (when the first operand entered into the pipeline arrives into the ring FIFO), the *DoneOperand* SR will point to this operand when the required *consuming* operand arrives.

2.5.3 Data Forwarding

Data forwarding is difficult in asynchronous pipelines. A data forwarding topology in a free-flow pipeline is shown in Figure 6.17.

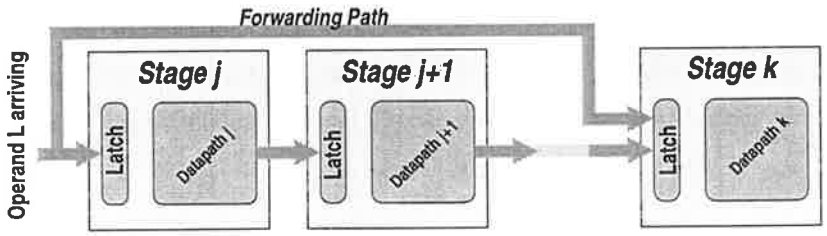


Figure 6.17 Forwarding Topology.

Data forwarding is much easier in synchronous design because the j^{th} stage and the k^{th} stage are in explicit synchronisation, and thus the only real issues are those to do with physical design, wiring delays and driving buffer sizing. However, in the free-flow case, the k^{th} stage cannot be said to be in synchronisation with the j^{th} stage, and indeed, the j^{th} stage may not even have the data that is required by the k^{th} stage yet. One solution to the problem is to enforce synchronisation across the part of the pipeline requiring a forward, which forces the j^{th} through k^{th} stages to operate simultaneously and only when all stages have received data. This may be acceptable for small $(k - j)$, but in the general case a better solution able to maintain the asynchronous nature of free-flow would be preferable. Although *halting* the free-flow pipeline until data is ready would seem another possibility, the halt will *thrash* because the latency of exiting the halt impacts on when the next operand for forwarding becomes available.

Data forwarding is a special case of a pipeline *fork*. A mechanism for data forwarding using a fork-style mechanism is shown in Figure 6.18.

The exact timing with which operands arrive at the *Producer*, the j^{th} stage, is unknown,

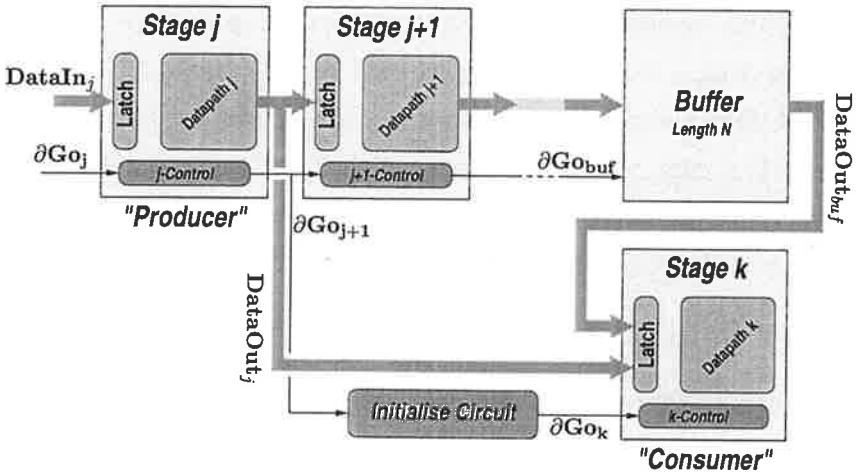


Figure 6.18 Free-Flow Forwarding Structure.

although the maximum rate is T_{issue}. Therefore, the forward path must *wait* until the data to forward becomes available, that is, arrives at the output of the j^{th} stage. The buffer is sized to hold the number of outstanding operands that can potentially be waiting for forward data to arrive, and is thus

$$N_{buffer} = (k - j) - 1$$

There can never be any *greater* overflow because the number of operands between the j^{th} and k^{th} stages is limited to N_{buffer}. Since this buffer does not require any special control and there are constraints on both the latency and bubble propagation time of the unit, a ring FIFO would best be used for this block. The *Initialise Circuit* does not pass the first $(k-j-1)$ ∂Go_{j+1} events, and then after every ∂Go_{j+1} produces a ∂Go_k event. This ensures that the k^{th} stage is only triggered when both the input and forwarded data are available, and this ensures that the k^{th} stage removes an item from the buffer (which has just been latched by the k^{th} stage).

3 Advanced Design Issues

The fundamental structures of free-flow pipelines have now been considered. These make the approach functional, but there are some further abilities desired. Asynchronous systems have the ability to change their operation rates in a data-dependent manner, which could result in significant performance gains in free-flow because of a lack of communication overheads. In addition, because free-flow is fundamentally an asynchronous approach, mixed synchronisation domains should not be discounted, requiring a method for interfacing free-flow and asynchronous systems.

3.1 Variable Latency Operation

Free-Flow pipelines can support variation of stage delays in the pipeline. The issuing stage and pipeline control can be designed such that stages may respond in differing, deterministic time to different operand types. Three methods for the implementation of variable stage latencies shall now be considered, called GREEN, BLUE, and RED.

The design of variable delay stages requires a good understanding and method for the design of delays and programmable delays. These requirements are discussed in Section 4.

3.1.1 GREEN Method

The GREEN method allows the operands to run essentially unchecked down the pipeline, with each stage controlling its own MDE element such that the stage delay guarantees that the stage completes the required operations before the ∂Go_{out} signal issues. The GREEN method is shown in Figure 6.19.

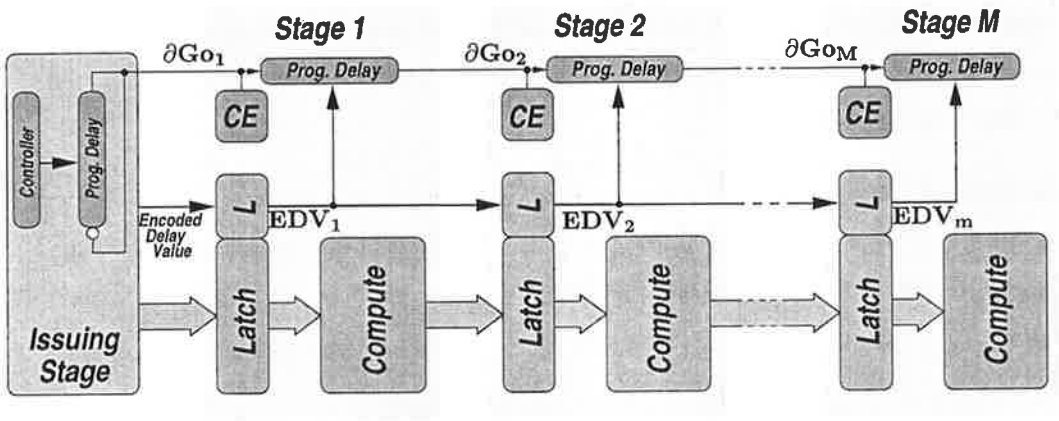


Figure 6.19 ***Green*** **Variable Delay Method.**

The ISS assigns each operand an *Encoded Delay Value* (EDV), that controls the programmable delay elements in each stage. Delay design is facilitated if a scheme called *row-hot* is used. In a *one-hot* scheme, a vector of width N is required to encode N different values, and only one bit of the vector is ever true. In a *row-hot* scheme, the vector width is still the same, but every bit *below* (less significant than) the encoded bit value in a *one-hot* scheme is also true. When designing *row-hot* codes for delay control, it is assumed that increasing the number of bits set *true* in the vector monotonically increases the delay provided by the programmable delay elements.

The GREEN method involves considerable complexity in the ISS in the general case. The ISS must determine before issuing the i^{th} instruction,

- the delay for the $i-1^{th}$ instruction to reach the end of the pipeline, $T_{to\,end\,i-1}$, and
- the delay for the i^{th} instruction to read the last stage, $T_{to\,N\,i}$.

The ISS thus issues the i^{th} operand a time

$$T_{issue\,of\,i} = max(T_{to\,end\,i-1} - T_{to\,N\,i}, T_{issue\,min})$$

where $T_{issue\,min}$ is the minimum issuing time of the ISS. This constraint ensures that operands do not collide as they traverse the pipeline logic. This will get unwieldy very quickly as the size of the pipeline grows and the number of possible variations between instructions grows, because the ISS must contain appropriate circuits and delays to track the timing between instructions as they move down the pipeline. For these reasons, the GREEN method is not considered in detail further.

3.1.2 BLUE Method

The BLUE method for variable latency implementation is shown in Figure 6.20. The BLUE method issues operands into the pipeline as soon as it can, but provides additional control so that the operands behave in the correct way as they propagate down the pipe. This eliminates the complex control of the *Green* method at the expense of global wiring and a rate that can be slightly below optimal.

The BLUE method will only allow the operand to be assigned *one* delay value as it propagates down the pipeline, and thus this assigned delay value should be the worst delay that this operand will experience in any one pipeline stage.

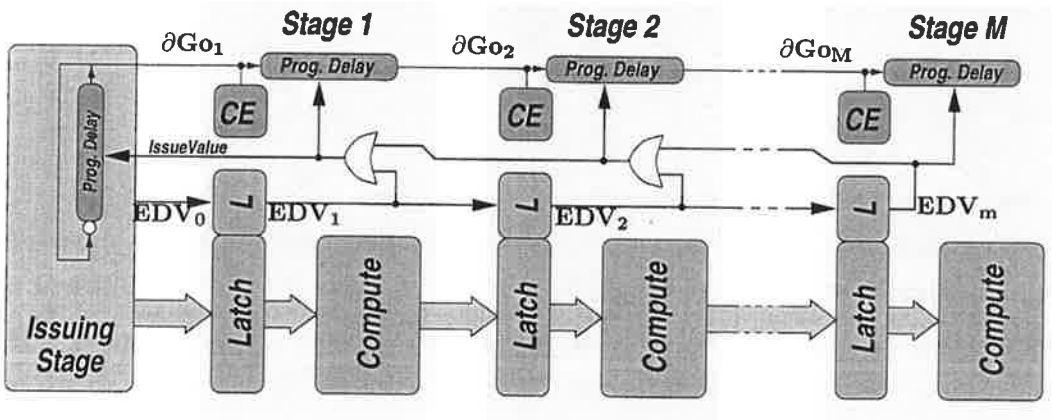


Figure 6.20 ***Blue* Variable Delay Method.**

The ISS assigns each operand an *Encoded Delay Value* (EDV) as it issues the operand into the pipeline, with the EDV value in *row-hot* form. This EDV value is latched and ORed with

the returning EDV value from the subsequent stage, and is then driven to the programmable delay element in each stage to control the stage latency. This ORed vector finally arrives at the ISS and controls the issuing rate for the next instruction, *IssueValue*.

When an operand with long latency enters the pipeline, the number of bits high in EDV_0 will be large, and this value will instantly feed back to the ISS through the first OR element. As this operand travels down the pipeline, with careful attention to EDV latching issues, the large EDV value continues to remain on the *IssueValue* bus, avoiding a long propagation path for the worst-case pipeline delay through the OR chain. When this operand leaves the pipeline, some parts of the EDV value will go to zero (if the operand following this worst operand has a lower latency), and start propagating back up the OR chain. The exact timing or how the delay elements are affected is not important, because the monotonic change in delay value with number of bits set high in the EDV value ensures that each stage and the ISS issue at or below the maximum rate permitted by the present state of the pipeline.

There is no hard limit on the length of the pipeline as the *row-hot* scheme coupled with the delay design ensures that long latency operands leaving the pipeline cause the pipeline to operate at or below the maximum issuing rate possible. This is slightly sub-optimal, but in short pipelines the performance lost will be negligible. However, the BLUE scheme does not allow delays to change dynamically as the operand progresses down the pipeline, since the ISS must make a decision at issue time about the latency of the operand in *every* stage of the pipe.

3.1.3 RED Method

The RED method is the most general, but the most complex, method for controlling variable-latency operations as they propagate down a free-flow pipeline. Two stages of the RED method are shown in Figure 6.21. In the RED method, the control will tolerate the pipeline stage latencies changing dynamically with operand type. This ability is what is ideally wanted from the pipeline control.

The issue with varying the delays in this manner is to ensure that each stage does not operate faster than successive stages can tolerate due to their own latencies. This is, in general, a difficult problem because the pipeline stages are essentially autonomous. In Figure 6.21, the delay value for a stage is computed *ahead* of time (in the previous stage) before being forwarded. This latched EDV_i value is ORed using a dynamic gate with the incoming value from the subsequent stage. The dynamic OR is used so that the EDV value that this stage experiences can be evaluated *and held* even if the subsequent stage becomes free (which would make determination of required delay difficult). Thus, once evaluated by stage i, the EDV_i value should *not* be precharged *nor* evaluated until the next operand arrives for this

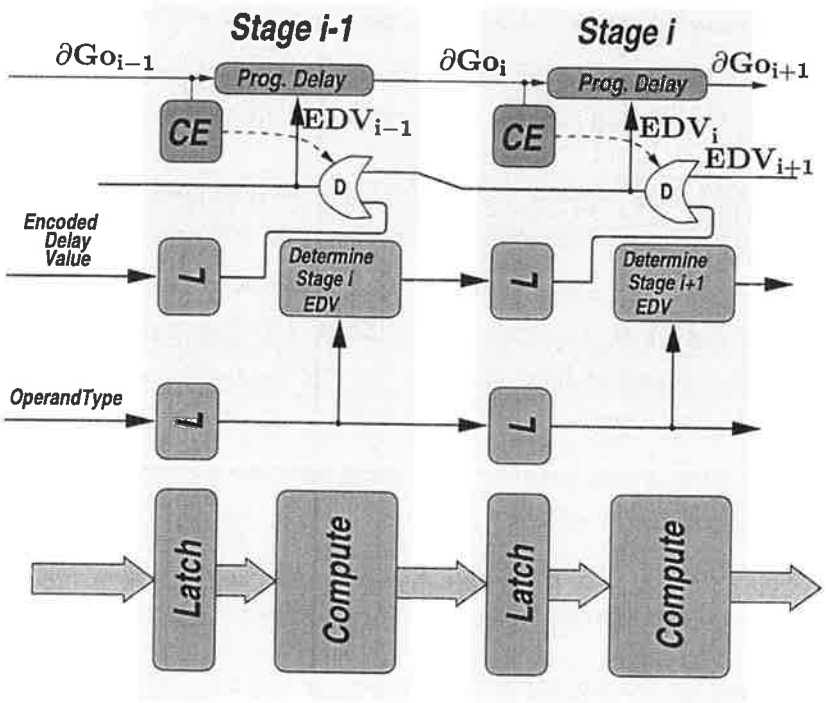


Figure 6.21 ***Red* Variable Delay Method.** This is one possible method for control of the pipeline under varying stage latency conditions. The *D* OR-gate is a special form of dynamic gate.

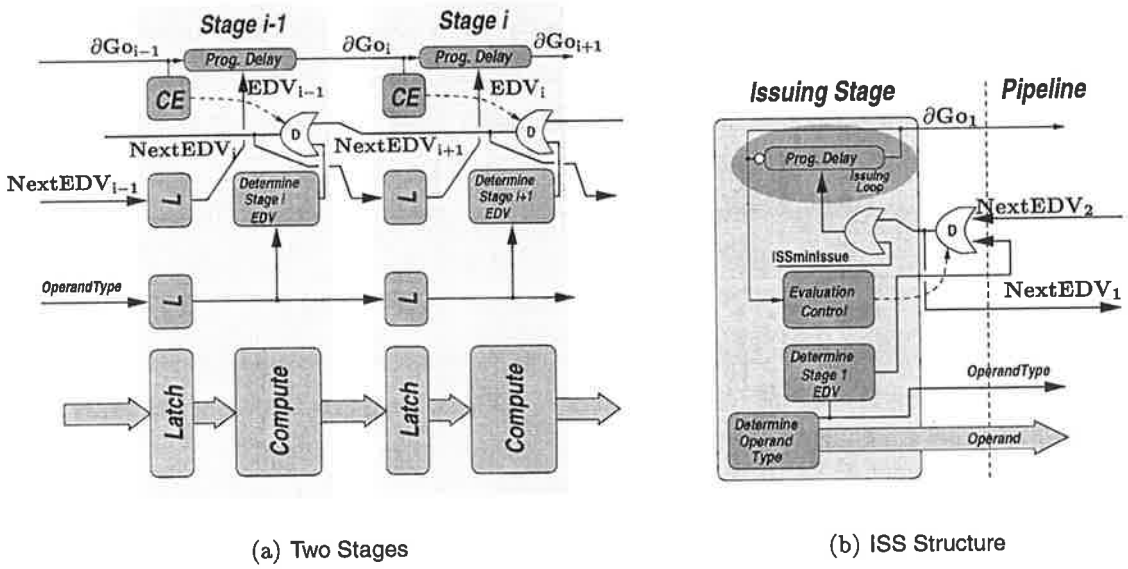
stage. Using a dynamic gate eliminates the latch that would otherwise be required. This also introduces an additional constraint. As the arriving EDV_{i+1} is expected to be valid, this stage must allocate adequate time for the subsequent stage to evaluate and propagate EDV_{i+1} to stage i before commencing it's own evaluation of a new operand. This places a constraint on the cycle time of the total system,

$$T_{issue} - T_{MDE\,i} \geq T_{ready\,edv_{i+1}} + T_{prop\,edv_{i+1}} - T_{need\,edv_i}$$

where $T_{ready\,edv_{i+1}}$ is the time after ∂Go_{i+1} that the $i+1^{th}$ stage produces EDV_{i+1}, and $T_{prop\,edv_{i+1}}$ is the time taken to propagate it to the i^{th} stage, and $T_{need\,edv_i}$ is the time after ∂Go_i that the i^{th} stage requires EDV_{i+1} to be valid. In a balanced pipeline stage, where $T_{MDE\,i} \approx T_{issue}$, some issuing rate *margin*, creating dead-time in each stage, is generally required to allow evaluation and propagation of the EDV values. This dead-time can also be used for precharging each stages' logic, thus suggesting that the RED method would well be used with dynamic logic datapaths.

It may be noted from Figure 6.21 that the evaluation of EDV_i is on the critical path because it affects the programmable MDE circuit. This would place an additional constraint on the minimum MDE value in each stage such that the EDV_i value was properly evaluated before the programmable part of the delay became active. This can be eliminated by *pushing* the

evaluation of EDV_i into the previous stage, shown in Figure 6.22, along with the structure of the ISS.



(a) Two Stages

(b) ISS Structure

Figure 6.22 Lookahead *Red* Variable Delay Method. The generation of the required MDE control values between stages is shown in (a), with the ISS stage in (b). The ISS evaluates its dynamic-OR gate when it *issues*.

The required minimum T_{MDEi} is now only the time between ∂Go_i and the inputs to the programmable elements becoming valid. In this case, the dynamic OR gates are evaluated *just before* issuing, such that the value they provide to the delay latch for the next stage (one of the $NextEDV_i$ values) is valid when latching occurs. The ISS, shown in Figure 6.22(b), evaluates the dynamic OR just before it issues, and this also ensures that the value into the ISS delay loop is static during the operation of the ISS. The dynamic OR output provides the delay value for the first stage ($NextEDV_1$) and is ORed with the minimum ISS issuing time to give the ISS latency on the next iteration.

Persistency and *Time-Out*

The special dynamic-OR gates used are *persistent* in that a long time after their evaluation, they will continue to hold the required EDV value even if this value does not mirror the present state of the pipeline (as it could be totally empty). Thus, an operand arriving will experience operand latencies that mirror the state of the pipeline after the last *operand* injection, without any consideration for the time that might have elapsed between these two events. Therefore, the $NextEDV$ values can be *timed-out* in the dynamic-OR gates, setting their outputs to zero when a *timing bubble* appears in the pipeline.

One approach would be to have an explicit global signal that *timed-out* all the $NextEDV$ dynamic gates after the ISS stalls — this works, but may not take into account fine variations in ISS timing that may introduce available timing bubbles into the pipeline. The detection of a appropriate timing bubble, which will cause an arriving operand to operate at its *inherent* rate instead of pipeline-determined rate, should ensure that the potentially *faster* operand always has an empty stage waiting for it, otherwise the rate constraint may be broken. Therefore, to *time-out* the dynamic-OR gate in the i^{th} stage, both the i^{th} and $i+1^{th}$ stages must be free. A signal derived from these two conditions may be used to generate an additional *reset* signal into the dynamic-OR gate in the i^{th} stage.

3.2 Interfacing

Interfacing free-flow systems to compatible domains is an important issue, especially since the possibility of *mixed* synchronisation domains or methods used in a system should not be discounted.

3.2.1 Free-Flow to Asynchronous Interfacing

The interface from an asynchronous pipeline to a free-flow pipeline must ensure that the rate of insertion of operands to the free-flow pipeline does not exceed the maximum permissible rate. One simple method of achieving this would be to set the cycle time of the last asynchronous stage to be the T_{cycle} of the free-flow pipeline. Another method is shown in Figure 6.23. This does not require modification to the asynchronous system, and simply forces the issuing rate of the ISS to be obeyed.

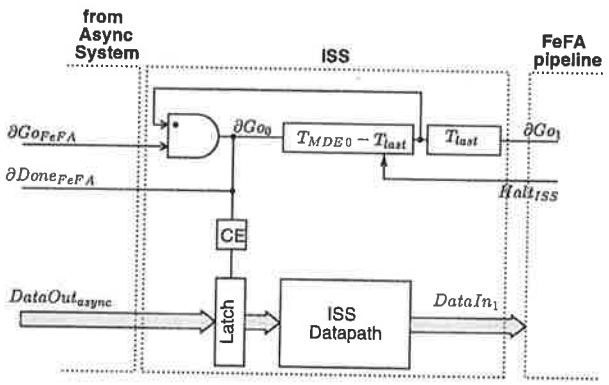


Figure 6.23 Asynchronous to Free-Flow Pipeline Interfacing.

The related problem of a free-flow pipeline feeding an asynchronous system can be dealt with in a number of ways. One is to insert a single-stage buffer and monitor its status to generate a halt signal. This technique is shown in Figure 6.24.

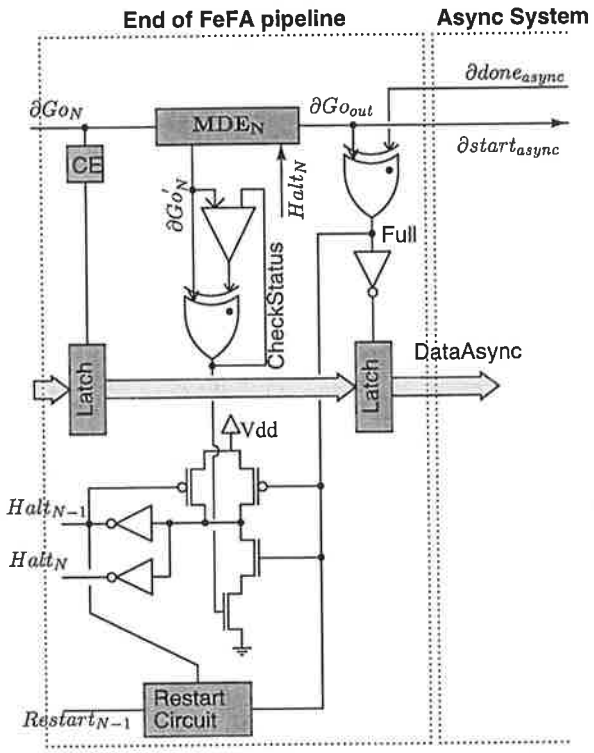


Figure 6.24 Free-Flow pipeline interface to asynchronous stages. The restart circuit will generate a low pulse on $Restart_{N-1}$ when the buffer stage empties.

The latch that generates $Data_{async}$ holds data for the asynchronous stage, and is required as the N^{th} stage can potentially receive data at the same time as sending it out. If the asynchronous stage fails to latch the data within a certain time (controlled by the *tap* in the delay MDE_N that generates $\partial Go'_N$), the $CheckStatus$ signal causes the halt circuit to be activated (since $Full$ is high as well), halting the free-flow portion of the system. When the asynchronous stage does return $\partial done_{async}$, the signal $Full$ resets, restarting the free-flow pipeline.

An alternative method is to *look ahead* into the state of the asynchronous pipeline to determine ahead of time if a bubble will arrive at the first stage *before* the next ∂Go_{out} event. This method is considered in more detail in Section 6.

If the free-flow pipeline is *not* haltable (for example, if a halt is simply *not* required in the pipeline), then the design of the asynchronous interface becomes more challenging. A simple solution is to adapt the interface of Figure 6.24 to the problem, for which a system topology like that shown in Figure 6.25 is assumed.

The $Halt_{N-1}$ signal is now sent directly to the ISS stage, however, the unhaltable system continues to run if any operands are in transit, and these operands should not be lost. If

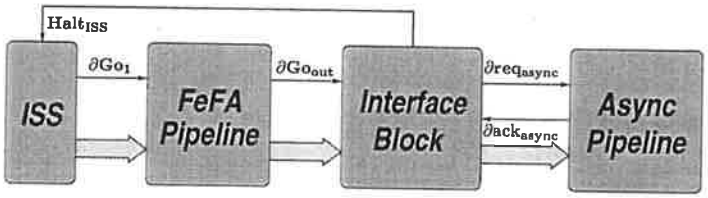


Figure 6.25 Interfacing an *unhaltable* Free-Flow pipeline.

the interface of Figure 6.24 is used, then a depth-M buffer will be required to absorb the operands, where M is the number of operands potentially in transit between ∂Go_1 and ∂Go_{out}. However, potential *thrashing* on $Halt_{ISS}$ when the asynchronous pipeline slows should be avoided. Therefore, a depth $(M+N_{buffer})$ can be used in place of the depth M buffer. When this buffer holds *nearly* N_{buffer} operands, the ISS must be halted. The exit of the halting condition (at what status of the buffer does the control restart the ISS) is then a design issue. This scheme for interfacing will be explored in greater detail in Chapter 7.

3.3 Overrunning the Pipe

As free-flow pipelines have no reverse-propagating control signals, they can be *overrun* — operated at rates higher than $T_{pl} + T_{data\,i}$. This requires careful attention to control issues and resulting constraints added to the system. Two possibilities are evident. One is to run the pipeline at a slightly higher rate than its normal supported rate, called a *moderately* overrun pipeline. The other is to operate the pipeline at a rate only determined by the characteristics of the combinatorial logic delays in each stage, called an *aggressively* overrun pipeline.

3.3.1 Moderate Overrun

When *moderately* over-running a free-flow pipeline, it is assumed that the requirements for the pipeline to *halt* (usually by the propagated mechanism) are still necessary, but some available timing margin is to be exploited by beginning a new operation in each stage earlier than that shown in Section 2.1.

The ability to overrun the pipeline and still require timing for halt propagation to be met requires that the i^{th} stage does not commence to send an operand to the $i+1^{th}$ stage until it is *sure* that no halting condition has arrived from the $i+1^{th}$ stage. This timing requirement is shown in Figure 6.26.

Therefore, a cycle time constraint in each stage is that

$$T_{cycle\,new\,i} \geq T_{post\,i} + T_{pre\,i+1} - T_{margin\,i+1} + T_{hp} + T_{setup}$$

where T_{setup} is the time required to setup the control input to the *send* gate after the $Halt_i$ signal actually arrives, and $T_{margin\,i+1}$ (defined in Section 2.3.2) indicates the timing margin

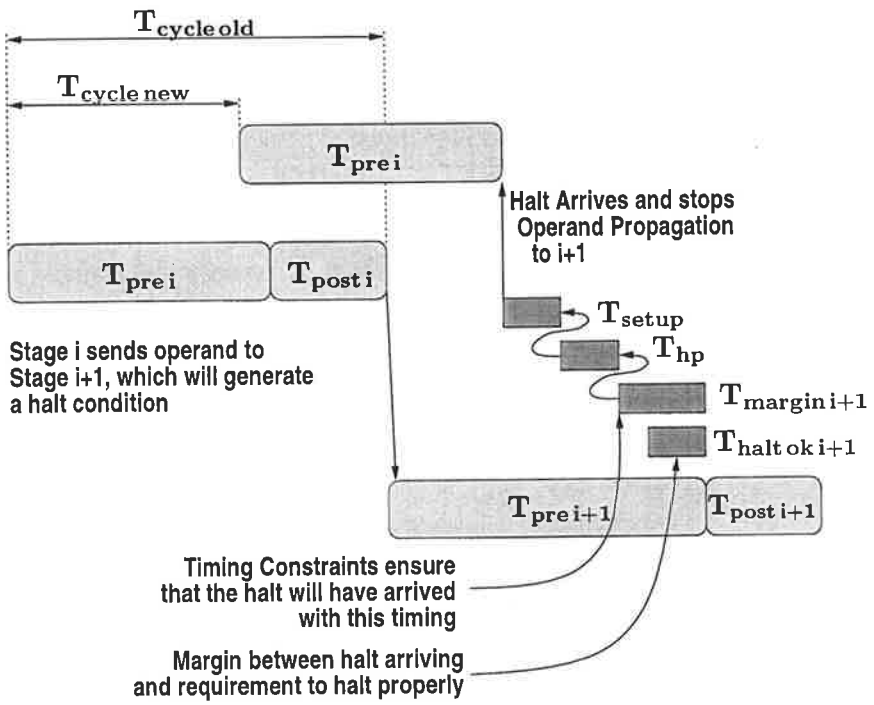


Figure 6.26 Moderate Overrun Halt Timing.

between the arrival of the halt signal and the time that the halt signal *must* have arrived to safely halt the stage. This constraint must be followed iteratively from the $N-1^{th}$ stage to the 1^{st} stage, and the peak allowable cycle time becomes the largest of the $T_{cycle\,new\,i}$ values. This makes halt timing constraints much tighter, since at some stages there will be close to *zero* margin between the setup of the halt and the arrival of the respective ∂Go event. Thus, a pipeline which already fully exploits the halt timing characteristic to construct a reasonably long pipeline will be unable to be over-run in this way.

3.3.2 Aggressive Overrun

The ability of logic stages to be clocked at a higher rate than their total propagation delay would allow has long been known [Cot65,Cot69,Kog81], and has resulted in a set of techniques currently known as *wave pipelining* [GLI94].

Free-Flow pipelines are amenable to wave pipelining techniques due to the absence of reverse-flowing control. It is impossible to wave pipeline an asynchronous pipeline stage (without the addition of extremely long latency buffers to catch any overflowing operands [SK96b]) because, by definition, only one operand may be propagating though the logic at any one time, which is *enforced* by the control protocol. However, free-flow lacks this limitation. Moving to a operand-flow methodology like wave pipelining demands that any ability to *halt* the pipeline be abandoned, since there will now be *multiple* operands traversing the logic of any one stage.

The logic timing diagram of a pipeline stage with *latches* at the inputs is shown in Figure 6.27. When the latch opens, new data starts propagating through the latches, although this data is assumed to be incorrect. Data at the inputs is assumed to go valid T_{pl} before the latch closes, and at this point begins propagating through the logic, with all outputs becoming valid $T_{slow\,i}$ later. The next stage then commences latching of this data, which must be held for T_{hold} in the i^{th} stage to ensure latching completes. A new data wavefront may then arrive.

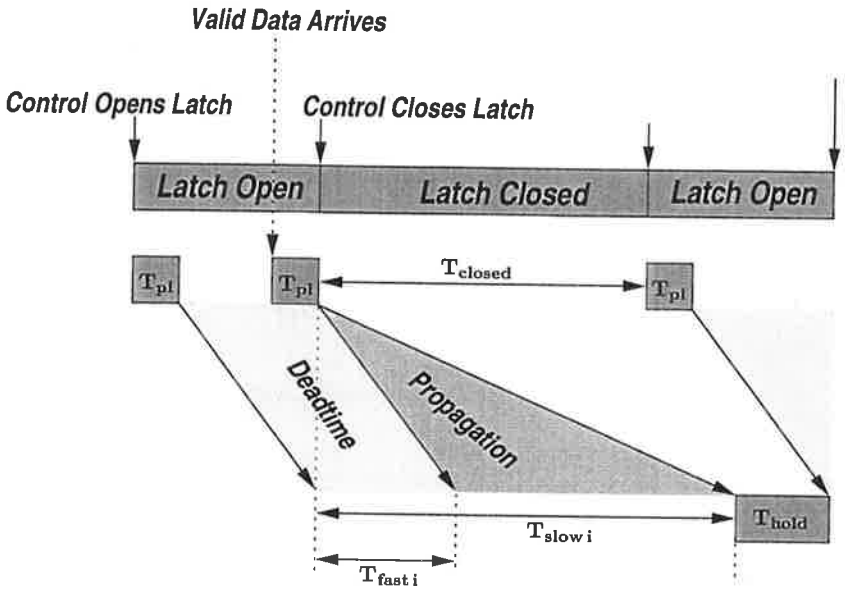


Figure 6.27 Pipeline *O verrun* Timing.

The challenge is to design the free-flow control around this timing to maximally exploit it. The structure of the IE controller (Figure 6.5(a)) does not require modification to support issue rates higher than that normally allowed. To meet the above timing requirements, the cycle time in any one stage is the maximum of two constraints,

$$
\begin{aligned}
T^1_{cycle} &= T_{slow\,i} + T'_{hold\,i} - (T_{fast\,i} + T_{pl}) + T_{send} + T_{until} + T_{\downarrow Lt_i} \\
T^2_{cycle} &= T_{until} + T_{\uparrow Lt_i} + T_{\downarrow Lt_i} + T_{send} \\
\Rightarrow T_{cycle} &\geq max(T^1_{cycle}, T^2_{cycle}) \\
&\text{where} \\
T'_{hold\,i} &= T_{hold} + (T_{MDE\,i} - (T_{pl} + T_{slow\,i}))
\end{aligned}
$$

T^1_{cycle} is a constraint that mirrors the requirements of the timing of Figure 6.27 such that the time the latch is closed is sufficient to ensure latching in the next stage is completed. T^2_{cycle} is the maximum rate supported by the latch control logic of the IE controller. The amount that any *overdesign* of the MDE element impacts on cycle time is factored in using the $T'_{hold\,i}$

parameter. Thus, the maximum achievable cycle time is approximately,

$$T_{cycle} \approx max(T_{slow\,i} - T_{fast\,i} + 2.7 \cdot \mathcal{T}_g, 3.1 \cdot \mathcal{T}_g)$$

where $\mathcal{T}_g$ is a gate delay in the chosen technology (see Appendix C). Therefore, even assuming *perfect* best and worst case path matching, the maximum achievable cycle time is approximately three gate delays on a per-stage basis. This limitation is caused by the IE controller keeping the latch open longer than T_{pl}, introducing deadtime into the logic path, which limits the peak achievable cycle time. The system cycle time will have to be set according to the worst-case stage.

IOE controllers

The structure of IE controllers is directly amenable to aggressive overrun because the latch control depends only upon the input event ∂Go_{in}. However, in IOE controllers the latch control depends on input to output event timing, and thus the delay element designs must be modified to support higher data rates.

The time taken to *open* the input latch in an IOE controller only depends on the first delay element, T_{d1}. This governs the T_{closed} expression (see Figure 6.27), and the value of T_{d1} gives a set of constraints,

$$\begin{aligned} T_{d1_i} &\geq T_{slow\,i} + T'_{hold\,i} - T_{fast\,i} - T_{pl} - T_{\downarrow Lt_i} + T_{\uparrow Lt_i} \\ &\text{and} \\ T_{cycle} &\geq T_{d1_i} + T_{\uparrow Lt_i} + T_{\downarrow Lt_i} + T_{pl} \\ \therefore T_{cycle} &\geq T_{slow\,i} + T'_{hold\,i} - T_{fast\,i} \end{aligned}$$

where $T'_{hold\,i}$ is identical to that for the IE controller. The IOE controller can thus achieve a cycle time of the theoretical maximum given the correct T_{d1} design, as it has the ability to minimise the *deadtime* of Figure 6.27 by only keeping the latch open for T_{pl}, which the IE controller cannot do.

Interfacing

At some point, this aggressively pipelined free-flow system will have to interface to another system, and since it lacks the ability to *halt*, some special structures will have to be used to ensure operands are not lost if a problem slows the receiving pipeline. These overrun pipelines are examples of *unhaltable* free-flow pipelines, in which a halt cannot be properly propagated to ensure no loss of data in the pipeline. Therefore, an interfacing scheme similar to that of Section 3.2 will must be used, which *catches* any overflowing operands in a buffer at the end of the pipeline. This buffer can be used to halt the stage which *issues* into the unhaltable section of the design.

4 Delay Design

Free-Flow pipelines make extensive use of delay elements of precise value to obtain performance from the asynchronous pipeline. However, delay design is rarely discussed in the realm of asynchronous methodologies. The choice of a signalling scheme which requires symmetric transition behaviour (since both edges of a transition are considered equivalent in terms of behaviour) also raises a number of issues in the design of delays suitable for use in free-flow pipelines.

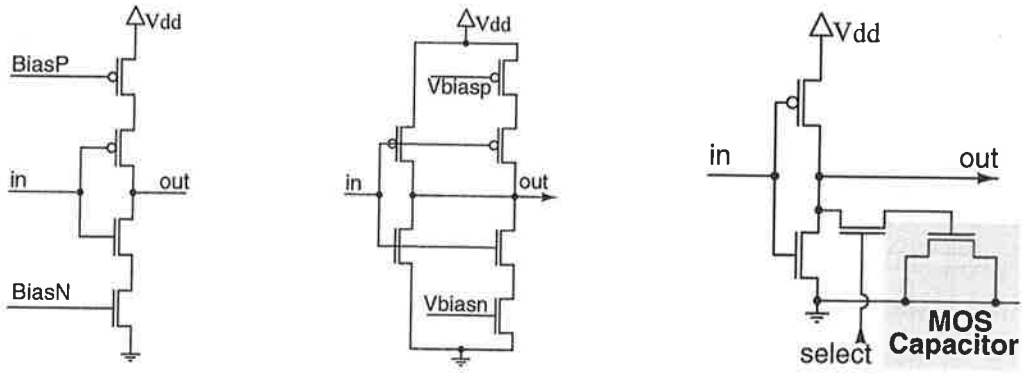
In Appendix A, additional data relating to delay design is presented.

4.1 Controllable, Programmable Delay Elements

The design of a stable and adjustable delay element is required for these pipelines (and 2ϕ ECS pipelines in general). The delay element should be programmable, either by digital or analog means, to enable the variable latency pipeline control schemes discussed in Section 3.1, and should attempt to minimise *skew* caused by mismatch between rise and fall times when driving a load. Several techniques for the design of such delay elements are shown in Figure 6.28.

4.1.1 Delay Implementation

There is a wide range of options for the circuit implementation of a programmable delay element. Several methods of implementing precise adjustable delays were examined [Dea92, MLCS93, Gra93], and three candidate circuits were identified out of the large number of circuits proposed. These in Figure 6.28.

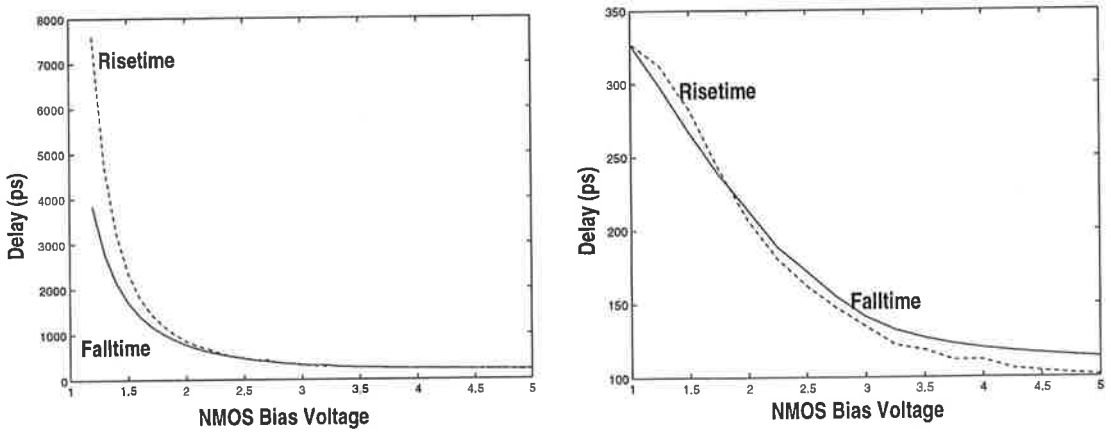


(a) Current-Starvation (b) Max+Current-Starvation (c) Switched Capacitor

Figure 6.28 Delay Element Implementations.

The *current-starvation* delay, Figure 6.28(a), uses bias voltages on transistors added to the output stacks that allow the amount of current available for load drive to be controlled. This results in a delay that is variable over a wide range. The delay element shown in Figure 6.28(b) is a modification of this to ensure that the *maximum* delay value produced by the circuit is bounded. The inverter that is not current starved sets this maximum value. The delay characteristics of these two elements are shown in Figure 6.29.

The *switched capacitor* element [Dea92] of Figure 6.28(c) is not suitable for subtle reasons. The output switching characteristic will always be slightly asymmetric because the gate capacitance C_{gs} of the MOS device (and thus MOS capacitor) is an asymmetric function of the applied gate voltage [Sch87]. Although this may be able to be somewhat mitigated (by using a full transmission gate and both a PMOS and NMOS load device), the skew between rising and falling edges is still problematic. In addition, if the control signal, *select*, is switched dynamically during operation the circuit ceases to operate as required. When *select* goes low, the charge on the MOS capacitor is stored, and when *select* goes high again, the output will either *glitch* or the delay will not mirror the fact that a capacitor has been switched onto the output.



(a) Current Starvation Curve

(b) Max+Current Starvation Curve

Figure 6.29 Biased Delay Element Characteristics. Note the wide delay range of the *current starvation* element, (a), which has a relatively flat slope for most ranges of bias voltages, followed by an extremely steep slope. The *max+current starvation* element, (b), achieves a better resolution over the entire bias voltage range. The PMOS device was biased at V_{dd} - $V_{bias\,N}$.

Although the *current starvation* element can achieve a wide delay range, it is very sensitive to bias voltage variations (see Figure 6.29(a)) and requires the generation of well-controlled

analog voltages. It is difficult to digitally program the element on a dynamic (highly time-varying) basis. However, the *max+current starvation* element achieves a much narrower and better defined dynamic range (see Figure 6.29(b)), and is suitable for digital programming because of the two *set* values at the extremities of bias voltage.

Integrating Delays

The integration of programmable delay elements with pipeline control logic is facilitated when using the *row-hot* coding scheme (see page 147). The interface between the pipeline latch and the delay elements is shown in Figure 6.30.

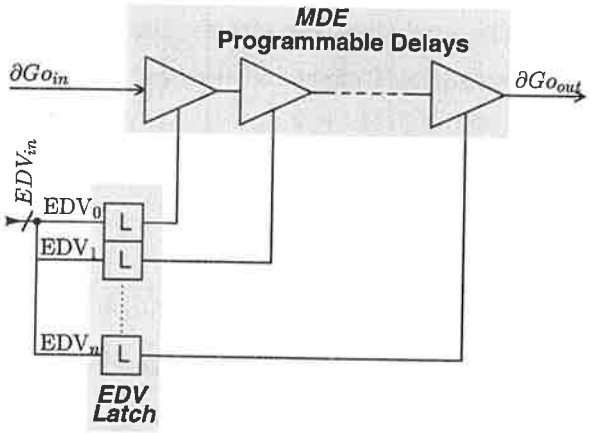


Figure 6.30 Pipeline Interface to Programmable Delay Elements.

If the *EDV* value were encoded (and used less than N bits to represent N differing values), then a decoder would be needed on the path between the latch output and the delay inputs. In addition, if variable latency were required, then a comparator (to decide the maximum value) and a decoder would be needed. This would then require a *minimum* delay value between ∂Go_{in} and the first of the programmable elements to ensure the controlling inputs were valid. Similarly, if a *one-hot* scheme were used, then a large *tree* of OR gates would be needed at the delay element control inputs to decide when to activate the relevant delays of the chain.

However, the *row-hot* scheme does not require any decoders before the programmable inputs, making the minimum required delay between ∂Go_{in} and the first delay element very low, and also making timing determinations for variable latency a single OR gate per *EDV* bit.

4.2 Delay Skew

The design of delay elements for use in free-flow pipelines, and in 2ϕ systems in general, is affected by a parameter termed *delay skew*, caused by asymmetric behaviour in the delay elements used in the control path. A simple representation of the forward control path in a free-flow pipeline is shown in Figure 6.31.

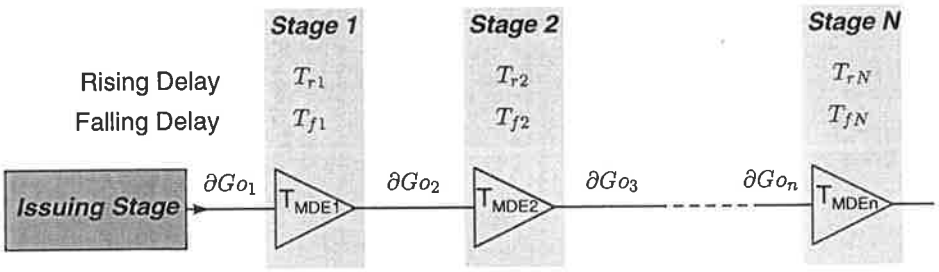


Figure 6.31 Forward Control Path in Free-Flow pipelines.

The delay experienced by an edge passing though the MDE varies depending on whether the input edge is rising or falling. The difference between rising and falling edges at the output of each MDE is termed *delay skew*, defined as

$$T_{ds\,i} = T_{r\,i} - T_{f\,i}$$

where $T_{r\,i}$ and $T_{f\,i}$, the rising and falling delays at the output of the delay element, respectively, are in response to a 50% duty-cycle waveform. The Matched Delay Element (MDE) value for this delay is defined as

$$T_{mde\,i} = \frac{T_{r\,i} + T_{f\,i}}{2}$$

These two definitions are shown in Figure 6.32.

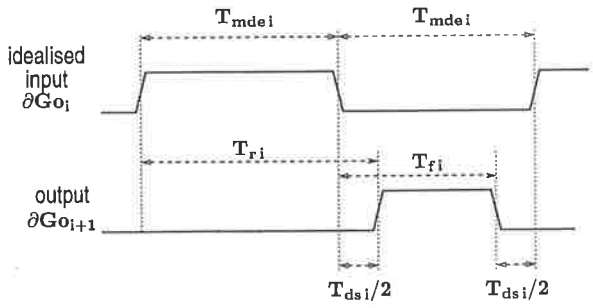


Figure 6.32 MDE Delay Skew.

If all the delay elements are non-inverting, then the difference between rising and falling delays in an MDE in each stage accumulate, causing the j^{th} stage to see a delay skew at the input,

$$T_{sds\,j} = \sum_{k=0}^{k<j} T_{ds\,k}$$

where $T_{sds\,j}$ refers to the summation of delay skews seen at the input of the i^{th} stage. This additive skew term is shown in Figure 6.33.

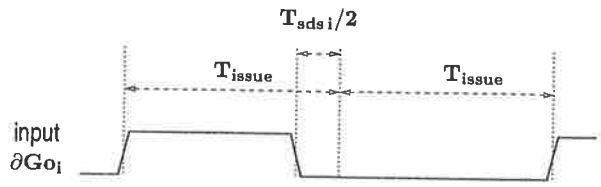


Figure 6.33 Additive Skew at Stage Inputs. The $T_{sds\,i}$ parameter represents the additive delay skew seen at the input of the i^{th} stage.

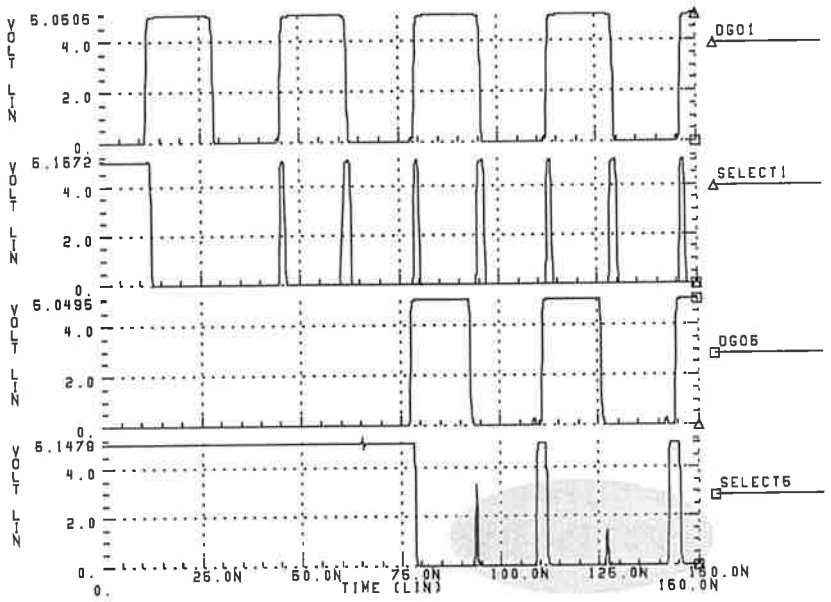
If this *delay skew* is not compensated for, it can cause failure in free-flow controllers. Since the IE controller only relies on the input ∂Go event, this controller will not fail in the datapath, but the computation path can fail because of inadequate computation time. In IOE controllers, the input is very sensitive to early ∂Go events (earlier than an issuing delay after the last operand arrived), and thus unchecked delay skew can cause total failure, illustrated in Figure 6.34 for an uncompensated and subsequently re-designed IOE controller.

One option is to simply allocate *margin* for delay skew in the ISS so that no subsequent stage can see any edge faster than its own cycle time. This causes a drop in performance (a problem in previous implementations [AML96]) which can be avoided if a way to mitigate the skew problem is found.

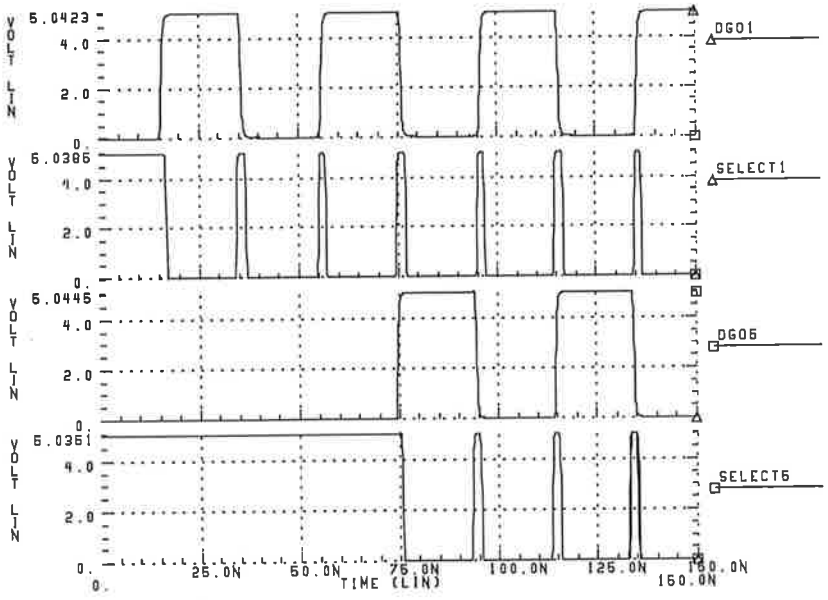
Definitions

If the added delay skews of each MDE in the pipeline are $T_{ds\,i}$, and each MDE is assigned a *sign bit* (if $\mathcal{S}_i = 1$ then the MDE is inverting, and if $\mathcal{S}_i = 0$ then the MDE is not inverting), then the general form for the delay skew is

$$T_{sds\,i} = T_{sds\,i-1} + T_{ds\,i-1} \cdot (-1)^{\left(\sum_{j=0}^{i-1} \mathcal{S}_j\right)} \tag{6.4}$$



(a) Unadjusted Free-Flow pipeline



(b) Compensated Free-Flow pipeline

Figure 6.34 Delay Skew Effects in Free-Flow pipelines. The latch control line in the 5^{th} stage, *select5*, has totally failed in (a). The compensated version using the *inversion elimination* technique eliminates the problem in (b).

where

$T_{sds\,0} = 0$ Zero Skew at "input" to ISS
$T_{ds\,0}$ skew added by ISS (see Figure 6.35)
$\mathcal{S}_0 = 0$ ISS defined as being non-inverting

The $T_{sds\,i}$ values represent the total *additive* delay skew seen at the input of the i^{th} stage, caused by the finite delay skews of the MDEs making up the pipeline control to this point. The skew generated by the ISS defined in Figure 6.35.

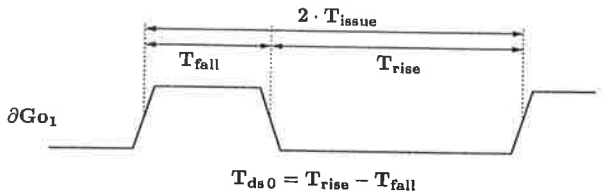


Figure 6.35 ISS added skew.

Inversion Elimination

If the required pipeline delays (MDEs) are all approximately equal, then each MDE can be made inverting, resulting in a delay skew

$$T_{sds\,i} = \begin{cases} T_{ds} & \text{if } i \text{ is odd} \\ 0 & \text{if } i \text{ is even} \end{cases}$$

where T_{ds} is the delay skew of each of the (identical) MDEs. This technique was used to compensate the control circuit whose response is shown in Figure 6.34. However, this technique is only applicable in a very specialised range of applications where the delays in each stage are not time varying and are virtually identical in every stage.

4.2.1 General Inversion Elimination

If the design of the required delays has been completed and the $T_{ds\,i}$ values are available, then a minimisation technique can be applied. If it is assumed that these $T_{ds\,i}$ values, implemented by MDEs, can be freely *inverted* and retain the same $T_{ds\,i}$, then the problem is then to find the $\mathcal{S}$ vector (of inversions) such that the effect of additive delay skew on the pipeline is minimised.

Minimisation Criteria

An initial optimisation criteria might be simply to minimise the maximum value of $T_{sds\,i}$ in the N-stage pipeline. However, this does not consider the actual effect of additive skew terms

on the pipeline. The real optimisation goal should be to minimise the *effect* of the $T_{sds\,i}$ values on the issuing rate of the pipeline. The additive skew terms' effect is such that

$$T'_{issue} = max(T_{sds\,1} + T_{mde\,1}, T_{sds\,2} + T_{mde\,2}, \ldots, T_{sds\,N} + T_{mde\,N}) + T_{margin}$$

The additive skew terms do not simply add to the issuing rate — the skew terms cause a localised derating of each stages' peak cycle time. Therefore, the issuing rate becomes the worst-case stage delay and additive skew term in the pipeline. The optimisation objective should thus be to minimise the worst-case value $T_{sds\,i} + T_{mde\,i}$.

The simplest method for this is a direct exploration of the complete space of the $\mathcal{S}$ vector. This is acceptable for typical length pipelines, however, this direct exploration algorithm does have exponential time complexity. However, extremely long pipelines would never be considered, since it is unlikely that this minimisation technique will result in optimal delay skew performance in practice.

4.2.2 Static Timing Skew Compensation (STSC)

Minimisation techniques based on nominal or worst-case skews from simulation can never be totally effective, because they do not account for temperature, voltage or process variation induced skews in the delay elements. The preferable solution is to eliminate delay skew to as great an extent as possible by using circuit techniques.

The *Static Timing Skew Compensation* approach (STSC) statically corrects gates for variances between pull-up and pull-down currents. If the circuit of Figure 6.29(b) is used, then the bias control circuit, shown in Figure 6.36 must ensure that the pull-down and pull-up currents are equal.

This circuit can also be used with the delay element of Figure 6.28(b), and has been used in the design of a fast clock reshaper circuit [MLCS93] and pattern generator [Moy96]. A large number of combinations of bias generator and delay element were tried before arriving at the bias control (Figure 6.36) and delay element (Figure 6.37), which produce excellent skew characteristics. The circuit also has the advantage of being *externally changeable* – if the delays are desired to be increased *globally*, then the $V_{mod\,p}$ voltage can be changed, and the bias point of the circuit changes appropriately to produce low skew in the delay elements.

Programmable Delay Elements

The bias control scheme can be used to ensure the path currents through the MOS trees are equal, however, a programmable delay element with minimum skew is still required. Figures 6.28(a) and 6.28(b) are combined to give the delay element of Figure 6.37(a). The

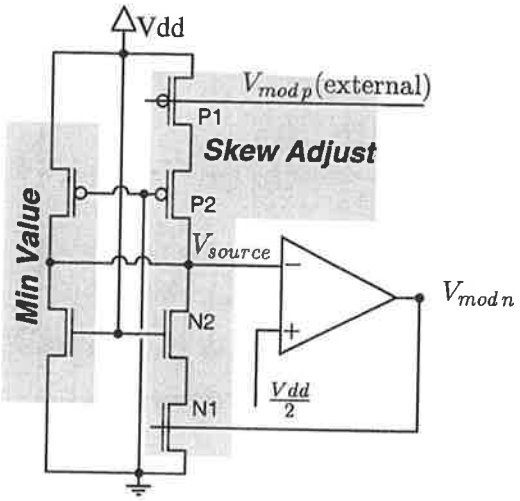


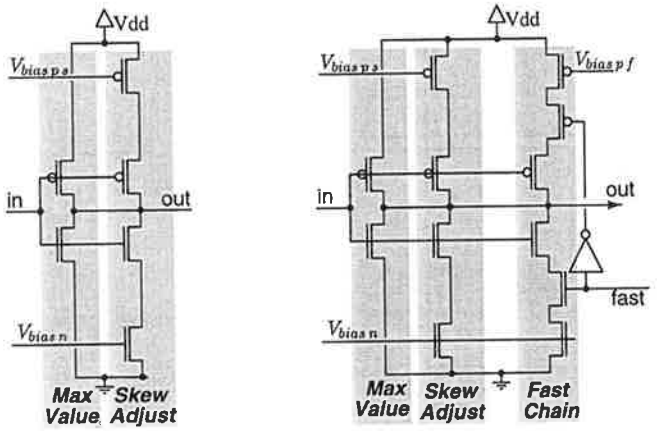
Figure 6.36 Bias Control Circuit for Tunable Delay Element.

bias voltages shown in Figure 6.28(a) are used to bias the individual elements for equal rise and fall times, with digital control being provided using the structure of Figure 6.37(b).

The *Skew Adjust* stack has a smaller $\frac{W}{L}$ ratio than the *Fast Chain*, so the circuit drives the node *out* in a time dependent on the state of the (digital) signal $fast$. The *Max Value* stack provides an upper bound on the delay value, with the PMOS and NMOS devices in this stack matched assuming nominal process conditions (maximising the ability of the bias circuit to correct any process-induced skew). The *Skew Adjust* stack corrects automatically for any imbalance in the *Max Value* stack, and including this stack (*Max Value*) improves the stability of the circuit greatly. Biasing for the two elements will use mirrored versions of each circuit of Figure 6.37 with a bias circuit of Figure 6.36 for each type of delay (thus, at least two bias circuits will be needed when using programmable elements). The skew performance of the STSC bias control with the delay elements shown in Figure 6.37 is shown in Figure 6.38.

Cancelling Offset

The STSC technique causes some finite timing skew offset in the delay elements because of offset voltage in the opamp of Figure 6.36. Fortunately, this will manifest itself in most delay elements almost identically as a finite skew offset in each delay. This effect can be cancelled to some extent by using the general *inversion elimination* technique, discussed in Section 4.2.1. The skew value introduced by each delay element in the pipeline due to some finite bias voltage offset is used as the $T_{ds\,i}$ delay skew value in each stage. However, not all of the skew introduced by the STSC technique corresponds to offset error, and is not avoidable by any other means.



(a) Basic Delay (b) Programmable Delay

Figure 6.37 STSC delay elements. (a) is the basic delay cell, which is mirrored and used with the STSC bias generator of Figure 6.36 to produce V_{biasps}. (b) is a programmable cell, again mirrored to produce V_{biaspf} using the STSC bias circuit.

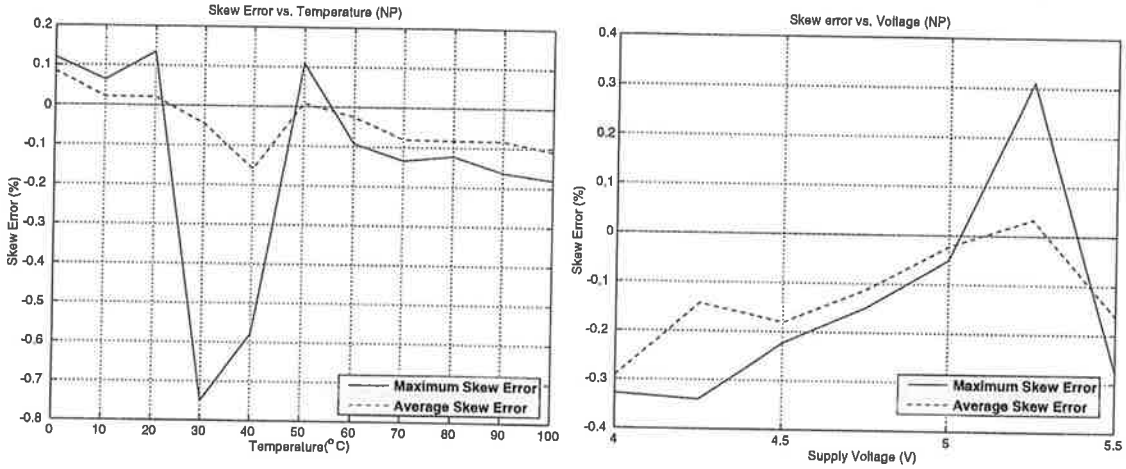
Skew Error Effects

It can be seen from Figure 6.38 that there is a small, non-negligible value of introduced skew due to temperature and voltage variation. This skew, in general, cannot be altogether cancelled by the general inversion elimination technique (Section 4.2.1) and some margin will have to be added to account for skew effects. The skew introduced by each delay element will be non-zero, and will require *derating* the cycle time considering raw delay element values. Two derating terms in each stage are defined — $\mathcal{K}_{T_i}$ and $\mathcal{K}_{V_i}$ for temperature and voltage effects, respectively, defined in each stage in general so that the characteristics of each stages' delay element may be considered. The uncancelable timing skew, $T'_{ds\,i}$, introduced in each stage is modelled by these two derating terms using

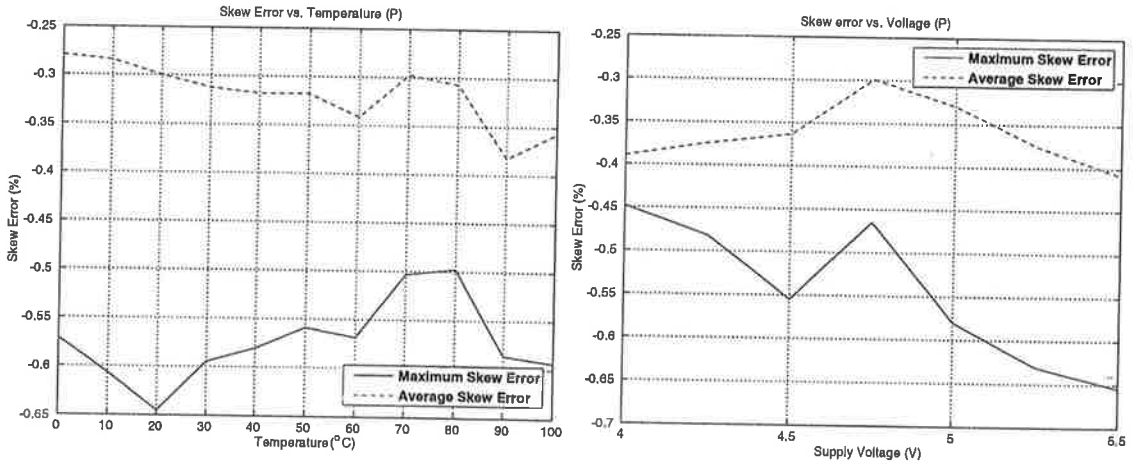
$$T'_{ds\,i} = T_{MDE\,i} \cdot (\mathcal{K}_{T_i} + \mathcal{K}_{V_i})$$

where $T'_{ds\,i}$ is the skew error in each stage, assumed proportional to the delay value, and there is considered to be no correlation between the two coefficients, and thus worst-case skew must be considered. Figure 6.38 would suggest $\mathcal{K}_T \cong 1.001 \leftrightarrow 1.008$ and $\mathcal{K}_V \cong 1.001 \leftrightarrow 1.006$, depending on the delay element and the ability to cancel static timing skew offset. This corresponds to a combined derating term of $1.002 \rightarrow 1.014$, or 0.2% to 1.4% derating of delay on a per-stage basis. As this skew can generally not be eliminated by any other means, it must be compensated by increasing the cycle time of the ISS,

$$T'_{issue} = max(T'_{sds\,i} + T_{mde\,i}) + T_{margin}$$



(a) Non-Programmable Element Skew Error



(b) Programmable Element Skew Error

Figure 6.38 STSC skew performance. (a) shows characteristics for the non-programmable (NP) element of Figure 6.37(a), while (b) shows characteristics for the programmable (P) element of Figure 6.37(b). The data was generated from HSPICE simulations of the delay element circuits. Eight skew values at each data point were generated, the average skew error at each data point being the average of these eight values.

where T'_{issue} is the derated cycle time of the ISS unit, and the $T'_{sds\,i}$ values are calculated using Equation 6.4 by inserting the $T'_{ds\,i}$ values. Thus, the longer the pipeline, the greater the combined derating factors and the greater the potential impact on derated cycle time, T'_{issue}. This may be in direct conflict with performance requirements should the system behaviour be amenable to high degrees of pipelining, possibly presenting a fundamental impasse. Fortunately, *Multi-Rate*, to be discussed in Section 6, solves these problems.

4.3 General Timing Skew Issues

The design of delay elements to achieve close to zero timing skew operation is one problem that can be solved by a combination of circuit techniques and cell selection. However, delay elements are not the sole elements of the main free-flow control path for IE or IOE controllers. In general, forward control paths for free-flow pipelines require *send* gates to halt the pipeline during a halting condition. Eliminating timing skew in *send* gates is difficult because the gate is a complex element and techniques similar to those used for delays are not as elegant because of the circuit complexity of the *send* element, which, in a delay sense, should be identical to the *datapath latch* (so that the delay in the control path only mirrors the datapath delay).

One solution that is applicable is *inversion elimination*, discussed previously. Although the delay elements in each stage will rarely be identical, the *send* gates used will almost always be identical, so inversion can be used to mitigate the effects of the skew that they introduce, especially since the transparent latches used for *send* gates tend to be inverting [SY92]. This simple solution only requires attention to initialisation issues in the circuit implementation.

5 System Test

Free-Flow complicates the issue of test even more than the test of normal asynchronous pipelines (see Chapter 4.2) as there is a lack of sequential behaviour in the control that facilitates test to some degree. However, the addition of *halt logic* (Section 2.3) enables the insertion of test functionality into the pipeline.

5.1 Functional Test

The structure of one possible free-flow functional test system is shown in Figure 6.39.

If the pipeline length normally disallows complete halt by the *propagation* method, then the pipeline can be slowed by asserting the signal $AddDelayToISS$, adding an extra delay, Ta, to the ISS issuing loop. If the pipeline can be halted normally using the *propagation* method, then this control can be omitted. At the end of the pipeline, the N^{th} stage is halted during

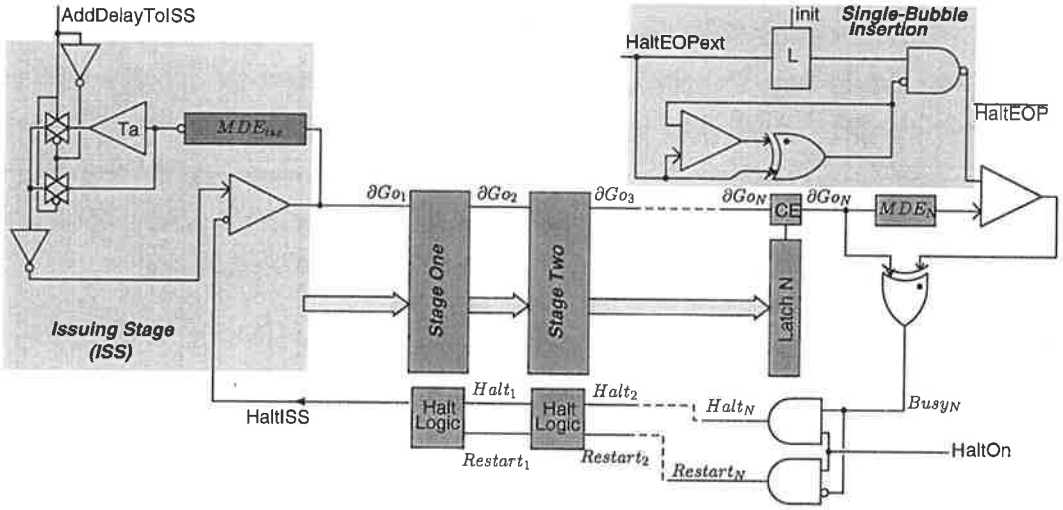


Figure 6.39 Free-Flow Functional Test Control. An addition to the ISS loop allows the pipeline to be *slowed* if required, while the circuit driving *HaltEOP* generates high pulses when halt is active.

test, allowing the pipeline to fill, after which scan can be commenced. The halt and restart functions are controlled by the two *and* gates, which generate a halt as soon as stage N becomes busy, and generate a *restart* signal when the stage becomes empty. The N^{th} stage is emptied by pulsing the $\overline{HaltEOP}$ signal, causing restart, which is generated by the circuit operating from $HaltEOPext$, an externally-applied signal. In this way, the free-flow pipeline can be single-stepped to facilitate scan test.

The structure of an individual free-flow stage facilitating scan test is shown in Figure 6.40. If an individual stage generates its own halting condition, then control similar to that of Figure 6.12 will have to be added to the stage.

5.2 Delay Test

The ripple mechanism described in Chapter 4.2.2 can be used to partially delay test the free-flow pipeline logic, which will require local control over the local scan registers in each stage with similar logic to control the latching of stage data when a single *bubble* is propagated back up the pipeline.

Another method that can be considered for delay test is *double-bubble* insertion. The schemes for test discussed so far insert a single *bubble*, or empty stage spacer, into the pipeline and then handle the resulting data generated from the propagation of this bubble. An alternative in delay testing is to insert *two* bubbles. The first bubble causes a computation at-speed, and the second bubble allows this result to propagate to the succeeding stage where it *halts*. The result can then be shifted out. The control necessary in the N^{th} pipeline stage (i.e. the end of the pipeline) to insert a double-bubble is shown in Figure 6.41. In this circuit, the

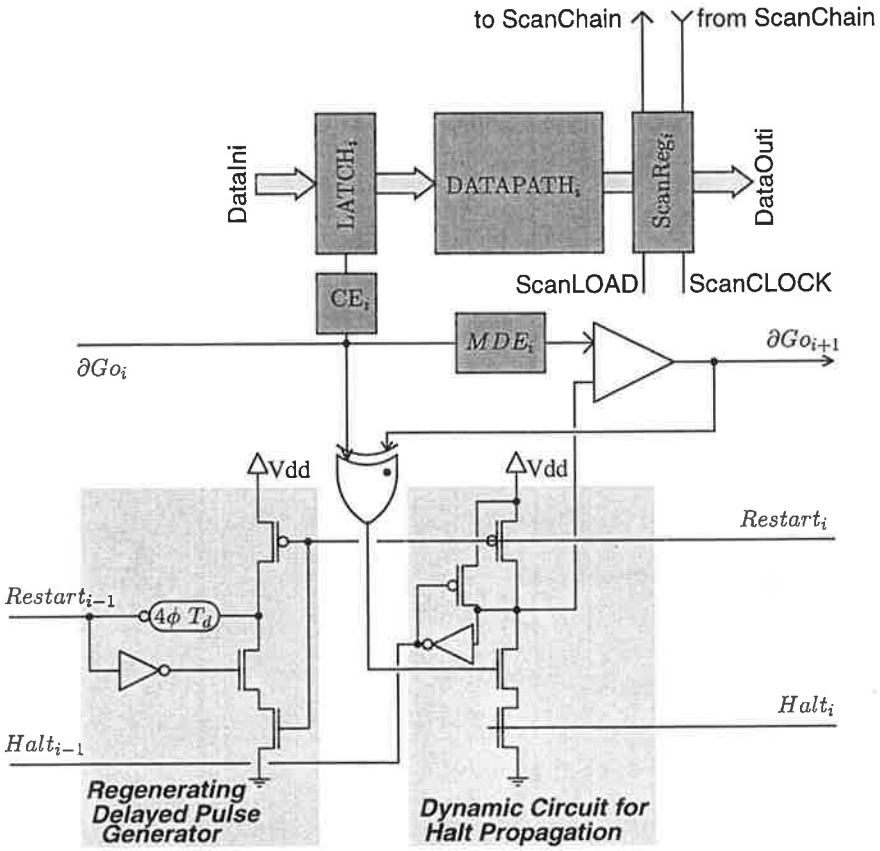


Figure 6.40 **Free-Flow Test Stage.** This assumes that stage *i* goes not need to assert its own halting condition, and simply propagates the incoming restart signal when ready. The regenerating circuit ensures that a good $Restart_{i-1}$ pulse is produced in case $\uparrow Restart_i$ occurs before the signal has propagated. This logic can also be used for *delay testing* of the pipeline.

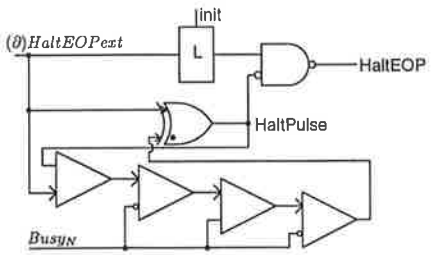


Figure 6.41 **Double Bubble Insertion.** This circuit replaces the single-bubble insertion circuit of Figure 6.39.

external signal $HaltEOPext$ is viewed as a semi-transitional signal. When the pipeline is in halt mode, a transition on $HaltEOPext$ generates a high pulse on $HaltPulse$, causing a high pulse on $HaltEOP$, inserting a bubble into the last stage of the pipeline. This causes the busy signal of the N^{th} stage to go through the sequence $0 \rightarrow 1 \rightarrow 0$ (inserting a double bubble), and the pulse is then removed, causing the next arriving ∂Go_N event to halt, and propagating a halt condition back up the pipeline.

A more complex scheme would be *triple-bubble* insertion. Double-bubble insertion brings every operand back to the bounded-delay constraint (because control and data are effectively synchronised at the stage inputs when the bubble arrives), which may cause some delay faults to be missed. Triple-bubble insertion would allow the *second* operand processed by the bubble insertion to both see the 'real' input control-data skew and operate at-speed. The control to achieve triple-bubble insertion is a straightforward extension to Figure 6.41, and requires no modification to the individual stage design, however only every third operand will be caught at the N^{th} stage for inspection.

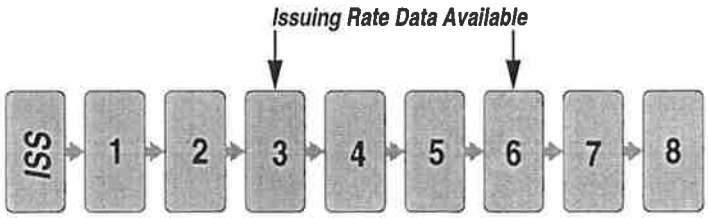
6 Multi-rate pipelines

In free-flow pipelines, the control may not allow optimal issuing rate decisions to be made in the first *conceptual* stage of the pipeline. However, the ability to make issuing rate decisions later in the pipeline may have considerable benefits in terms of throughput and latency. To overcome this limitation, the *conceptual* pipeline, consisting of many stages, in which some stages have the potential to make optimal or good issuing rate decisions based on supplied data, is split into shorter pipelines and connected with *buffers*. The resultant pipeline structure, shown in Figure 6.42, is called *multi-rate* (MR), since individual sections of the pipeline can now operate at different cycle times, as opposed to the worst-case cycle time of the entire pipeline, decided in the first pipeline stage of Figure 6.42(a).

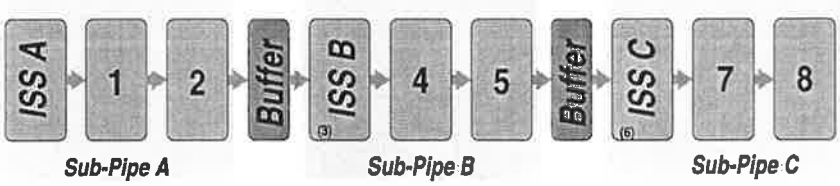
The Merits of Multi-Rate Free-Flow Pipelines

Multi-rate pipelines, as their name suggests, allow autonomous parts of a larger pipelined system to operate at their optimal rates without requiring consideration of interactions with other decoupled pipelines. Even though this may mean the pipelines may operate at varying rates, the expectation is that the architecture be designed so as to make the average rates of all pipelines approximately equal, thus maximising utilisation of the system.

Two other subtle features combine to make multi-rate systems highly desirable even if the pipeline tends to operate at fixed rates. Decoupling allows smaller sections of the pipeline to



(a) *Conceptual* Pipeline



(b) *Split* Multi-Rate Pipeline

Figure 6.42 Multi-Rate Pipeline Splitting. In the conceptual pipeline, (a), data on which to base a cycle time decision may not be available until later stages – the *ISS* must then assume the worst and issue at the worst-case rate of any of the stages. Splitting the pipeline and decoupling with short *buffers*, (b), removes this restriction.

assert halting conditions safely, due to the reduced sub-section pipeline length. In addition, the shorter pipes result in much lower values of non-cancelable timing skew (see Section 4.2.2), allowing the required derating of ISS cycle time to be minimised. The buffers allow the sub-pipelines to pass data only, and control information is not passed to the next pipeline, thus the *skew* present at the output of the sub-pipeline is not transferred through the buffer. Thus, the combined derating factor over the length of the shorter sub-pipelines will be significantly lower than the total derating factor of the pipeline considered as a whole.

6.1 Decoupling

The decoupling of free-flow pipeline sections from one another uses *buffers*. The interface at the end of the pipeline to a buffer is shown in Figure 6.43.

The buffer is assumed to supply two signals back to the control, *BufferFull* and *BufferRestart*. *BufferFull* goes high when the buffer is full, and should occur *immediately* after the last occurring ∂Go_{out}. *BufferRestart* indicates that space is available in the buffer, going high sometime after *BufferFull* makes a $0 \rightarrow 1 \rightarrow 0$ transition to indicate that the pipeline can be restarted. These two signals are used to control the *halt* condition within the pipeline, shown in Figure 6.44.

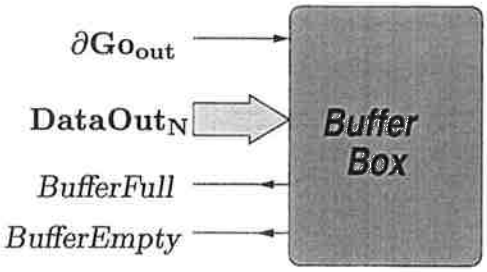
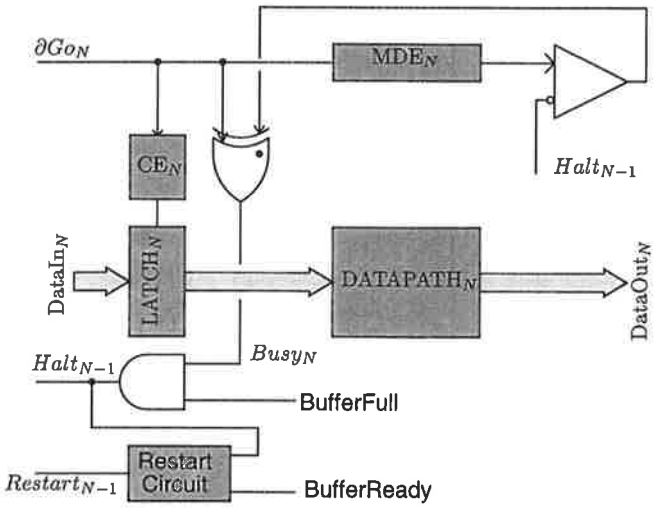
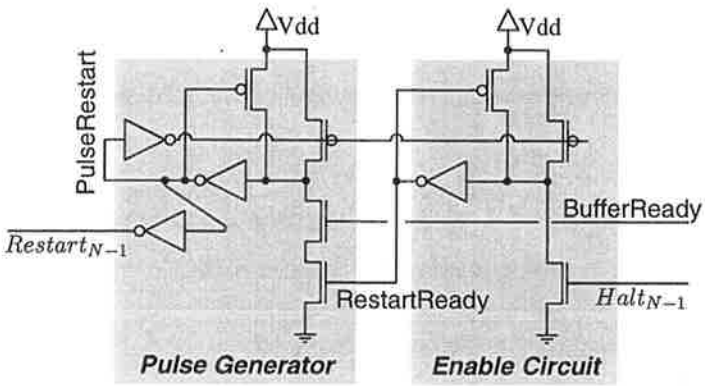


Figure 6.43 End-Of-Pipe (EOP) interface to Buffer.



(a) End of Pipe Free-Flow logic



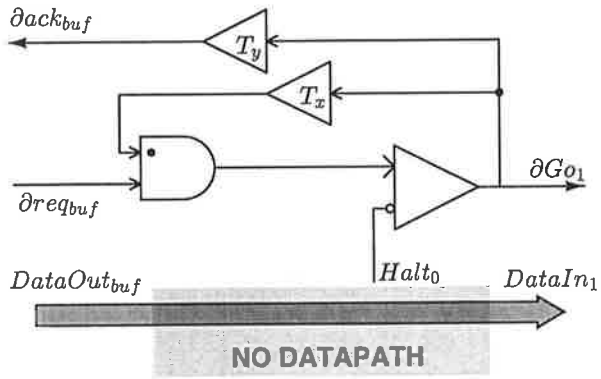
(b) Restart Circuit

Figure 6.44 Free-Flow End-Of-Pipe Control.

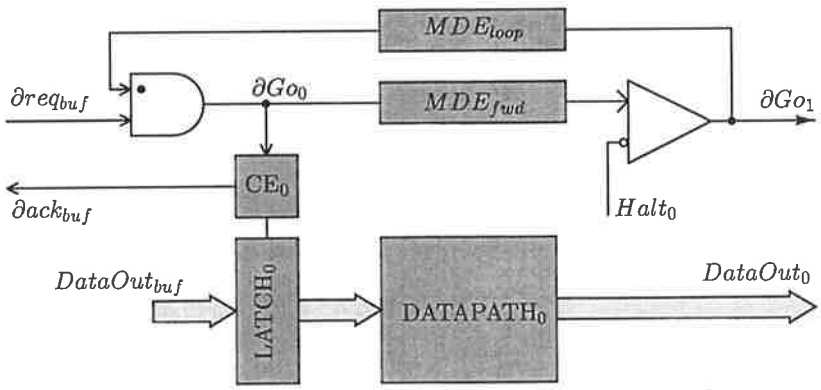
6.2 Multi-Rate Buffers

The implementation of the buffer to pipeline interface is dependent on the type of buffer used. These buffers will be some flavour of FIFO, and existing 2ϕ buffer designs (for example, the S-Pipe) can be used, or a ring-buffer type FIFO (see Figure 6.15, page 143) can be used to improve the latency of the buffer.

6.2.1 Two-Phase FIFO interfacing



(a) Simple 2ϕ interface



(b) Full 2ϕ interface

Figure 6.45 2ϕ FIFO interfaces.

A simple interface which implements a control loop that issues an operand to the first stage of the pipeline (via ∂Go_1) is shown in Figure 6.45(a), using the loop delay implemented using MDE_{fwd}. The ISS does nothing other than issue operands out of the buffer. However, if the

output of the buffer will be used to make issuing rate decisions (as would be expected in a multi-rate pipeline), then the circuit of Figure 6.45(b) should be used. The forward delay, MDE_{fwd}, accounts for the forward delay of the datapath logic in the ISS (here called the 0^{th} stage) before issuing to ∂Go_1 (with appropriate halt control added), and the return loop delay MDE_{loop} adds any return delay needed. The delay of the *last* gate merging the ∂req_{buf} and loop-return events can be pushed into the forward control delay with appropriate design of the control element, CE_0.

The end-of-pipe connection to the buffer connects $\partial req_{buf\,in}$ directly to ∂Go_N. The generation of the *BufferFull* signal requires attention to the bubble propagation speed in the asynchronous FIFO. The obvious way to generate the *BufferFull* signal is to AND the *Busy* signals in the 2ϕ FIFO structure. If the busy *vector* is denoted $\mathcal{B}$ (where $\mathcal{B}_j = 1$ if the j^{th} stage is busy), then the *BufferFull* signal can be generated via

$$BufferFull = \prod_{j=1}^{j=D} \mathcal{B}_j$$

where D is the *depth* to which the buffer is examined for empty stages (so that an incoming ∂Go_{out} will be received correctly). Consider the case in which an operand has just been inserted and the buffering FIFO is such that

$$\mathcal{B}_j = \begin{cases} 1 & \text{for } j = 1 \ldots D-1 \\ 0 & \text{for } j = D \end{cases}$$

If the bubble propagation speed from one stage to the previous stage is T_{bp}, then the depth D must be such that

$$D \leq \left\lfloor \frac{T_{cycle}}{T_{bp}} + 1 \right\rfloor$$

which is a *conservative* constraint as it ensures that the bubble has propagated from the D^{th} to the 1^{st} stage of the FIFO by the time the next ∂Go_{out} arrives. Generating the *BufferRestart* signal is trivial, and is a tradeoff between the frequency of halts in the pipeline and the dead-time produced by a halt. One possible expression for generating *BufferRestart* is

$$BufferRestart = \prod_{j=1}^{j=X} \overline{\mathcal{B}_j}$$

The most aggressive restart condition is $X = 1$ (with only one stage guaranteed to be empty), and the most *lazy* condition is $X = N$, where N is the depth of the FIFO, which waits until the pipeline is completely empty before restarting.

The choice of X would depend on the application. If the bandwidth of the *sending* system were slightly higher than the *receiving* system, then X would be closer to D. If the receiving system were slightly faster than the sending system, then starting the sending system as soon as possible would be advantageous, so X would be closer to 1.

Why Use 2ϕ FIFOs?

Two-phase FIFOs fit in well with the asynchronous concept of free-flow pipelines, and are very easy to connect to these pipelines for decoupling purposes. In addition, the generation of the *BufferFull* and *BufferRestart* signals is simple, being just an AND and NOR, respectively, of the pipeline *Busy* signals.

The control constraints on the operation of the FIFO are easily understood, and the structure is *neat*. In addition, some light *pre-processing* can be inserted into the FIFO with only a minor latency impact. However, when using a 2ϕ FIFO buffer there is a significant tradeoff between latency and ability to absorb variation. If the forward latency of the FIFO is T_{fwd}, then the total latency through the unit is $N \cdot T_{fwd}$. This may be unacceptable in many applications where the data from one pipe should arrive as soon as possible at the next, forcing the FIFO depth to be reasonably small.

6.2.2 Four-Phase Ring FIFO interfacing

A ring-buffer FIFO approach can improve latency as data does not have to propagate through the N-latches of an N-stage S-Pipe FIFO, instead propagating through only one. The ring buffer interface circuit to a free-flow pipeline is shown in Figure 6.46.

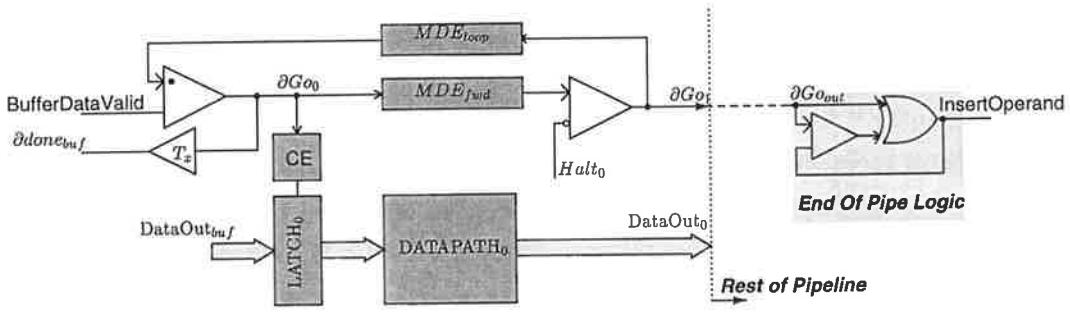


Figure 6.46 4ϕ Ring FIFO Interfacing. The interface to the buffer supplying data is a hybrid — the buffer indicates valid data ready with the logical signal *BufferDataValid*, and the interface returns the event $\partial done_{buf}$ as an acknowledge. The *End-Of-Pipe* logic sends operands into the ring buffer at the end of this pipeline.

The ring FIFO (shown previously in Figure 6.15) has very low latency (only one latch delay) when the buffer is empty. The ring FIFO interface, shown in Figure 6.46, assumes that the FIFO implements some additional functionality over that of the basic element shown in Figure 6.15. The FIFO is assumed to output a signal *BufferDataValid* to indicate the validity of the data at the inputs to this issuing stage. When the control detects this condition, it starts processing via the ∂Go_0 signal, returning an event $\partial done_{buf}$ to the FIFO, which should

advance the state of one of the shift registers (via *DoneOperand* of Figure 6.15) and annul the *BufferDataValid* signal for this part of the FIFO. The control then goes through the loop as normal, allocating a delay MDE_{loop} for any required ISS deadtime.

Why Use 4ϕ Ring FIFOs?

The ring FIFO structure significantly improves upon the latency of a 2ϕ FIFO implemented using a structure like the S-Pipe. This is highly advantageous as the normal pipe-to-pipe delay should be very low except when the receive pipeline halts. When the send and receive pipelines operate at roughly the same rate, then the latency through the FIFO stays at one stage delay, and the FIFO only starts to fill when the receive system slows or *halts*, allowing adequate timing margin in the sender pipeline for halting to occur.

It is difficult to add any processing to the ring FIFO due to its structure and very low nominal latency. However, the function of the inter-pipe buffer is to decouple pipelines with minimal latency impact, and not to provide pre-processing. Ring FIFOs are ideal for this purpose. In addition, a *kill* feature can be added to the ring FIFO (that eliminates all buffer contents) during special pipeline conditions, which will be advantageous when it is known that the contents of the ring FIFO are no longer useful (for example, if the FIFO contains instructions that are annulled during a mispredicted branch in a microprocessor).

7 Conclusion

This chapter has presented the Free-Flow approach (FeFA), from the basic conceptualisation of the idea and why it is necessary, through a complete spectrum of techniques necessary to make the design of free-flow systems easier and faster. Such techniques allow systems of a wide range of complexities to be implemented. FeFA provides a mechanism for significantly improving the performance of bounded-delay asynchronous systems.

The design of *delays*, central to the free-flow approach, has been carefully explored. Delay design for asynchronous systems is rarely examined seriously, however the requirement that free-flow introduces for tight, well-controlled and symmetric delays demanded a full investigation of the topic. Any sub-optimal delay performance (manifesting as *timing skew*) impacts directly on the cycle time achievable by the system.

7.1 Comparisons & Related Work

7.1.1 Synchronous Styles

The relationship between FeFA and synchronous design has already been explored, however there are a few additional points worthy of mention now that the complete approach has been described.

STRiP

STRiP [Dea92] uses a similar approach to the RED method for variable delay control detailed in Section 3.1, where the latency of the *next* worst-case operation is determined ahead of time and used to set the cycle time of the *next* operand. However, STRiP is a synchronous approach, requiring global clock distribution, even though it improves performance over a synchronous implementation by exploiting gross variations in pipeline latencies.

General Synchronous Design

Free-Flow has a similar communication overhead to synchronous styles (approximately zero) since hand-in and hand-out of data can be co-incident in any one stage. It is interesting to note that *both* the Free-Flow and synchronous approach suffer a similar problem at high speeds in a given technology due to timing asymmetries, which manifest themselves differently in each approach. In synchronous design, timing asymmetries in the delays through various paths of the global clock network result in small levels of uncertainty in the arrival times of the clock at the local latches. This *clock skew* causes cycle time degradation. In Free-Flow, *timing skew* results from edge-rate redistribution as edges pass through delay elements which are not totally symmetric, and these delay elements sequence inter-stage communications in the time domain. This effect, too, causes cycle time degradation.

Pipelined or *Buffered* clocking has been proposed for large linear arrays of processors [FK85] and systolic array processors [Kun88], where logic switching speeds are high and interconnect speed is low. Pipelined clocking sends the clock with data, and the receiver *buffers* the clock, sending it on when it has completed computation. This bears some similarity with the forward event path of free-flow, however, the clock period cannot vary in pipelined clock networks, nor can the clock be stopped. Interestingly, pipelined clocking networks must make allowance for the variance between rising and falling edges of the clock between processors [DS91], a situation virtually identical to the problem of *delay skew* in free-flow. This presents a fundamental limit on the speed and size of pipelined clocked systems, however, multi-rate is valuable technique that almost completely circumvents this problem in free-flow.

Free-Flow cannot accomplish some tasks, like data backwarding and forwarding, with the relative ease and lack of expense in hardware that synchronous design makes possible. This is unfortunate, and more work must be done on the *dataflow* aspects of the schemes to determine whether special techniques can be used to improve the structure of these problematic data-movement mechanisms.

7.1.2 Asynchronous Styles

Free-Flow significantly improves upon the latency and throughput of the fastest of fully asynchronous pipelines using a bounded-delay approach (the S-Pipe — see Table 4.1). This

is achieved while retaining the locality of asynchronous control, and retaining transparent power-down, haltability, and variable-latency operation.

However, Free-Flow loses the resilience of asynchronous approaches. A bad module in an asynchronous system will not cause total system failure, but the same cannot be said for a free-flow system because of the *set* timing requirement of the ISS. This perhaps is the price paid for extracting higher performance from the asynchronous paradigm.

Free-Flow does introduce timing issues into the design of logic designed to *halt* the pipeline, due to non-sequential control flow. This makes the halting of very long pipelines, in general, quite difficult without small added timing margins. This problem can be compensated for by using short multi-rate pipelines, which eases timing constraints on the halt logic, at the cost of a small increase in latency in the pipeline.

The area penalty imposed by using free-flow over asynchronous methods like ECS is expected to be minimal, since the datapath structures are essentially identical (using single-rail bounded-delay implementations), and only the control is structured fundamentally in a different way. There will be some area penalty when implementing techniques like STSC (due to the need for precise analog circuit elements like operational amplifiers), but this should be minimal in the context of the complete implementation.

7.2 Closing Remarks

Free-Flow is a new approach for asynchronous control of pipelined systems. It retains all the features of asynchronous design that were found desirable in the implementation of ECSTAC — locality in signalling and timing verification, automatic power-down, aggressive 2ϕ signalling, and locally controlled and timed blocks.

Free-Flow takes the performance goal of asynchronous systems using the ECS method to its logical conclusion. It is interesting to compare the approach taken in FeFA with system architecture philosophy [PH96], *make the common case fast, and the infrequent cases correct.* FeFA is an asynchronous control method that makes the common case *as fast as possible* when the pipeline is operating in steady-state with no operands causing system halt. Control features necessary in system design are not sacrificed, even though they may be used infrequently.

Chapter 7

Free-Flow Communications

FREE–FLOW systems were originally conceived to solve throughput problems in asynchronous communication links. Asynchronous communication, although highly tolerant of timing variations and changing environmental conditions, can suffer performance degradation when the latency between the sender and receiver of data is relatively large compared to the speed of the sender and receiver circuits, since round-trip delays factor into every exchange of data. FeFA eliminates the acknowledge and replaces it with a relatively rarely-used *halt* signal, allowing a great deal of implementation-specific timing information to be used to determine how fast to operate the channel. This concept, *Non-Acknowledging Communication* (NAC), the precursor of the free-flow approach, is the subject of this chapter.

Many communicating systems are wire constrained, especially those that communicate between chips or boards because wiring is not free or always routable. An asynchronous approach that uses the same mechanism for long-latency off-chip communication as short-latency inter-module communication on-chip will invariably create restrictive bandwidth constraints on the communication channel. NAC is an asynchronous communication mechanism that does not restrict bandwidth through signalling convention, or require excessive wiring, and will be compatible with two-phase ECS systems (or general asynchronous systems), as well as free-flow implementations.

1 Asynchronous Channel Communication

The structure of an asynchronous communication channel is shown in Figure 7.1. Such a channel could be between communicating systems on a single chip (relatively short latency),

between chips on a PCB (long latency), or between boards on a backplane (very long latency).

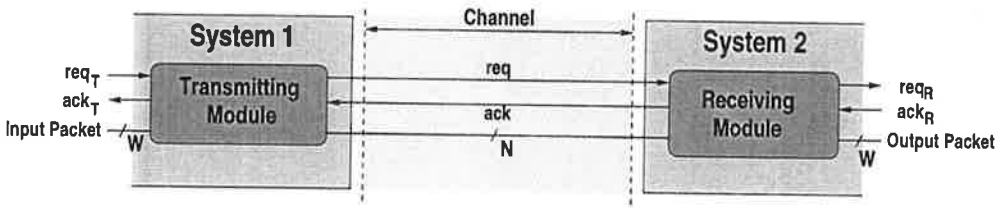


Figure 7.1 Asynchronous Communication Channel.

The asynchronous channel *transmitter* receives a request req_T, to transmit a W-bit packet of data across the channel, to which it sends ack_T if it is not busy. The transmitter has access to a N-bit channel for the transmission of the packet, where $W \geq N$. It does this by using the channels *req* and *ack* wires to sequence the communication. The receiver, upon the full W-bit packet arriving, sends the packet out via req_R and ack_R.

The performance of the channel depends on two factors — the scheme used to sequence the flow of data across the channel, and the width N of the channel itself.

Channel Timing

Timing parameters associated with the various parts of the transmission and reception system shown in Figure 7.1, defined so that the relative performance of various schemes may be examined, are shown in Table 7.1.

T_{cl}	Time to traverse channel from System One to Two
T_{ci}	Time to inject new operand into pipelined channel
T_{cr}	Time for System Two to acknowledge channel *req* with *ack*
T_{cnr}	Time to inject new operand from receipt of System Two *ack*
T_{cn}	Time to free transmitter from *ack* of last word, get new word and start transmitting

Table 7.1 Channel Timing Parameters.

A parameter M, representing the number of N-bit transfers necessary to transfer the W-bit packet, is defined, where

$$M = \left\lceil \frac{W}{N} \right\rceil$$

1.1 Fully Asynchronous Link

A fully asynchronous link would demand that every N bit block sent over the channel be acknowledged, and thus round-trip delays are factored in to *every* N-bit packet transmission. Throughput and latency for this scheme are

$$T_{throu-a} = M \cdot (2 \cdot T_{cl} + T_{cr}) + T_{cnr} \cdot (M-1) + T_{cn} \quad (7.1)$$

$$T_{lat-a} = T_{cl} + (M-1) \cdot (T_{cr} + 2 \cdot T_{cl} + T_{cnr}) \quad (7.2)$$

where throughput is defined as the time taken to send one complete *word* as seen by the interface to the transmitter, and latency is defined as the time from receipt of the W-bit packet to the last N-bit packet arriving at the receiver. The asynchronous transmitter demands an *ack* for every N-bit packet transmitted, and this transmission takes $2 \cdot T_{cl} + T_{cr}$ each time. Upon the receipt of the *ack*, the transmitter takes T_{cnr} to inject a new N-bit packet to the channel — this occurs $M-1$ times, until the last packet when the transmitter *frees*, taking T_{cn} to receive a new W-bit word and start sending.

1.2 Modified Asynchronous Link

Since the transmitter knows that the receiver is prepared to accept a complete W-bit packet, the sending of each N-bit packet need not be acknowledged, instead only requiring that the complete W-bit transfer be acknowledged [YHJN95]. This scheme will achieve a higher channel utilisation as it can take advantage of channel pipelining. The throughput and latency of this approach will be,

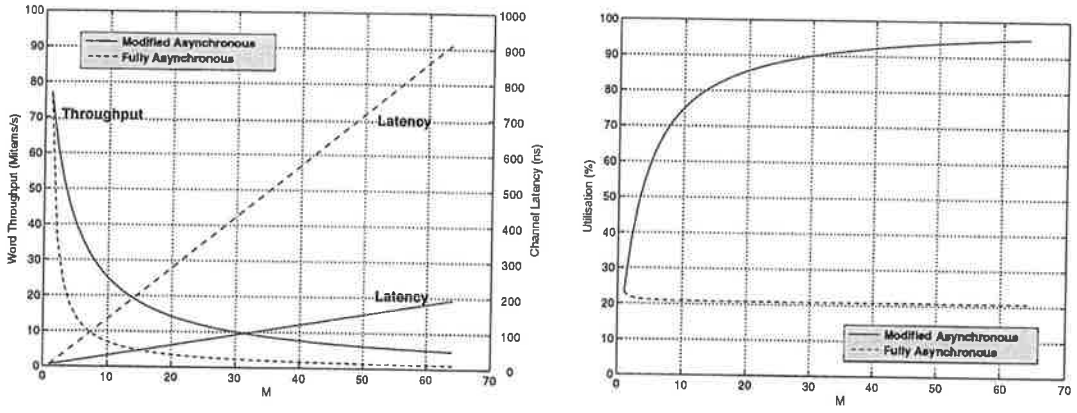
$$T_{throu-ma} = (M-1) \cdot T_{ci} + 2 \cdot T_{cl} + T_{cr} + T_{cn} \quad (7.3)$$

$$T_{lat-ma} = T_{cl} + (M-1) \cdot T_{ci} \quad (7.4)$$

The throughput of this scheme is only limited by the speed at which $M-1$ packets can be injected into the channel (at a rate governed by T_{ci}), and then the M^{th} packet acknowledged ($2 \cdot T_{cl} + T_{cr}$ after transmission of the last packet).

The performance of these two schemes is shown in Figure 7.2, with timing parameters characteristic of a 0.8μm CMOS process inserted into Equations 7.1-7.4. A standard CMOS pad model was used, assuming the channel latency, T_{cl}, is 5ns, with a T_{ci} of 3ns. The channel return time, T_{cr}, was 1.5ns corresponding to the $\partial req \rightarrow \partial ack$ delay of the S-Pipe, also setting the time to free the transmitter, T_{cn}, to 1.5ns. A delay of 3ns was used for T_{cnr}.

Utilisation in Figure 7.2(b) is defined as the throughput divided by the *potential* throughput of the channel if a perfect transmitter and receiver are used (set by T_{ci}). The latency of both channel architectures grows linearly with M, the number of transfers needed to send



(a) Throughput & Latency

(b) Utilisation

Figure 7.2 Asynchronous Communication Link Performance.

the complete W-bit packet. Throughput similarly drops with increasing M. However, the channel utilisation of the modified asynchronous scheme improves with M — this is because the portion of time spent in control tasks decreases as M increases. However, the setting of M to give a satisfactory utilisation may not give a reasonable throughput. An option for this scheme is to place a *buffer* in the receiver, so that the sending of one packet occurs in parallel with the acknowledge of the last packet, which should result in higher throughput.

2 Free-Flow Communication Architecture

Applying free-flow concepts to the design of the channel interface can simultaneously give high throughput and utilisation, due to the lack of required acknowledges. A potentially pipelined channel of the type shown in Figure 7.1 may be considered as an *unhaltable* free-flow pipeline. The structure of the free-flow interface to the channel is shown in Figure 7.3.

The interface is similar to that developed in Chapter 6.3.2. Each operand is *re-assembled* after being transmitted through the channel, and then enters a L-place buffer before being issued to the receiving system. The *Buffer Monitor* observes the status of the buffer, sending the *Halt* signal to the transmitting system if the space available in the buffer is only just sufficient to hold the potential number of outstanding operands both in the channel and the *operand assembly* stage.

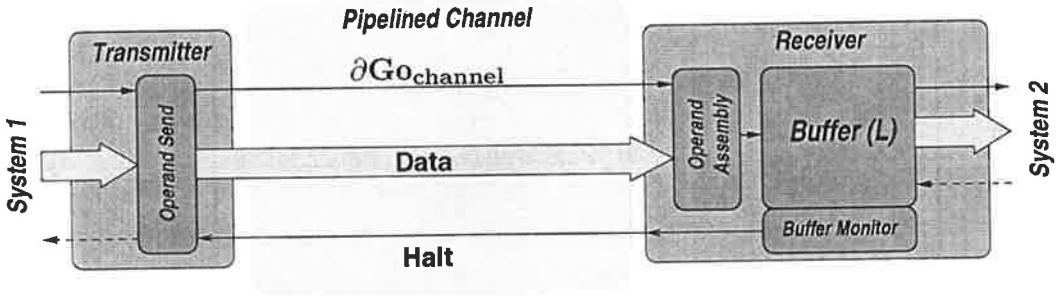


Figure 7.3 Free-Flow Communication Channel. The dotted lines to both System 1 and 2 indicate that they may be either asynchronous or *free-flow*. The *Halt* signal is only asserted when the buffer only has space left for the items potentially in transit between the sender and receiver.

2.1 Buffer Sizing and Halt Assertion

The size of the buffer, L, must be set such that the worst-case conditions in both transmitter and receiver do not result in the loss of operands sent between the two systems. The timing parameters of interest are illustrated in Figure 7.4.

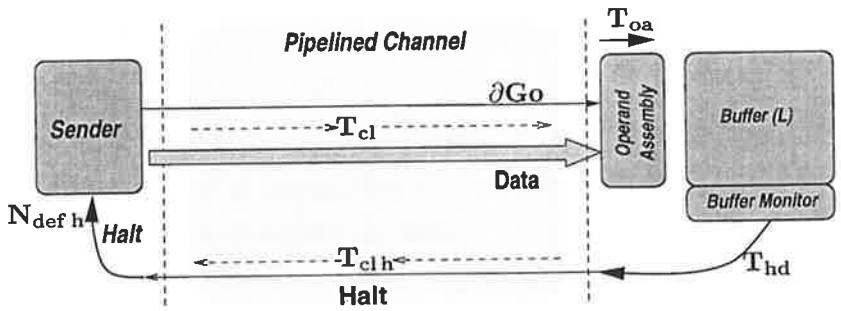


Figure 7.4 Timing Parameters for Buffer Sizing.

T_{cl} is the latency of transmission through the channel, while $T_{cl\,h}$ is the latency of transmission of the *halt* signal, which may be different from T_{cl} if a different signalling scheme is adopted. T_{oa} is the *Operand Assembly* block latency, from the time the block receives the last packet necessary to output the word until it actually produces an input to the *buffer*. T_{hd} is the time from the insertion of the relevant operand into the buffer until the detection of the halt condition. $N_{def\,h}$ is the number of input operands (at the system side) for which the sender can *potentially* defer the taking of the action associated with the arrival of the halt. T_{ci} is the channel injection time, the required time spacing between data packets injected into the channel interconnect.

The size of the buffer is chosen such that

$$L = N_{def\,h} + \left\lceil \frac{T_{cl} + T_{oa} + T_{hd} + T_{cl\,hh}}{M \cdot T_{ci}} \right\rceil + N_{extra}$$

The fraction represents the total number of *whole* packets that may be in transit after detection of the halt until the halt arrives at the *sender* side. The upper integer bound is used in this expression to gain the worst-case figure, since it is assumed that the sender will *not* stop sending parts of an unfinished word transmission, and halts when a new packet is received from the system. However, the system may *defer* the processing of the halt action for a number of received input packets (if a metastability-resilient circuit is used that trades time for failure rate), and this worst-case number $N_{def\,h}$ is added for safety. $N_{def\,h}$ would normally be at least one, to represent when the halt *just arrives* when a new packet is received from the system (the minimum value of $N_{def\,h}$ may well be zero from timing considerations, but using a minimum of one is reasonable).

The term N_{extra} sets the amount of *slack* in the buffer before a halt must be asserted to ensure the system halts safely and data is not lost. With $N_{extra} = 0$, a halt would have to be asserted as soon as the first packet arrived (so as not to lose any packets potentially in-flight), so some non-zero value of N_{extra} is required to minimise *thrashing*.

2.2 Merits of NAC

The latency of transmission of NAC is similar to that for the modified asynchronous scheme, since the time required to send the M packets is identical in both cases. However, because NAC is a free-flow technique, it does not require an acknowledge unless the receiving system slows in its processing of transmitted packets, and can start sending the next packet as soon as the sending of the last word is completed. This eliminates the bandwidth restriction imposed by the modified asynchronous scheme, which required an acknowledge before the next packet was sent, and thus throughput is potentially

$$T_{throu-NAC} = M \cdot T_{ci}$$

The ability of the NAC technique to approach this upper bound will depend on the quality of the circuit implementation.

3 Channel Control

In this section, the interfacing of NAC circuits to asynchronous modules at the sender and receiver side of the channel will be considered. This will involve the design of control functions

and circuits necessary to enable the operation of the NAC scheme. However, detailed circuits and schemes for high-speed I/O signalling will not be considered, a separate topic in itself [CCS+88, JK88, Røi96]. The interfacing of this channel to a free-flow system is a simple extension to the architectures described.

3.1 Full-Width *NAC* channel

The first design considered is relatively simple in that the width of the channel is equal to the width of the incoming packet to be transmitted. This simplifies the design of the sender and transmitter circuits considerably.

3.1.1 Sender Control

The simple sender interface for the $M = 1$ channel (in which W, the width of the packet to be transmitted, equals the channel width N) is shown in Figure 7.5. The sender receives a W-bit packet from the asynchronous system, driving it into the channel along with the $\partial Go_{channel}$ event. However, the input *send* gate is only enabled when *HaltSender* is low, and this *send* gate is a point of potential failure due to metastability.

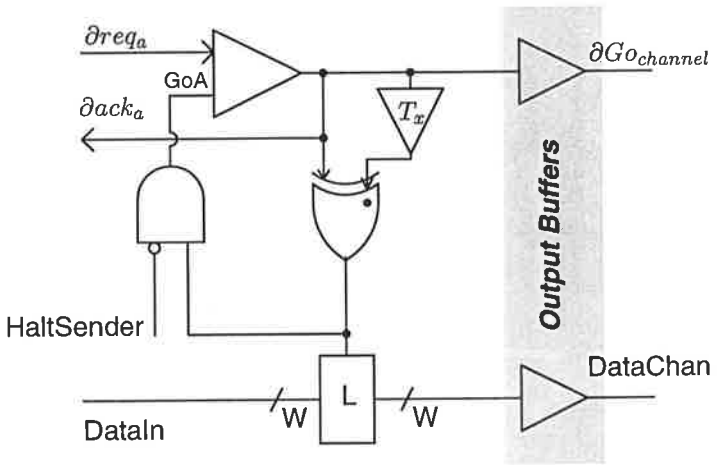


Figure 7.5 NAC Sender Control for M=1.

The delay T_x resets the sender once a period has elapsed that ensures proper data and control edge injection into the channel. T_x is implementation-dependent. If the required *spacer time* in the channel is T_{ci} (as used before), then the delay T_x is

$$T_x = T_{ci} - (T_{until} + T_{\uparrow Lt} + T_{and} + T_{send})$$

assuming that the fast-path and response constraints (see page 60) for this pseudo-S-Pipe stage are met.

Mechanics of Halt Control

The exact timing relation between the sender and receiver is unknown, because both are asynchronous systems. The *send* gate of Figure 7.5 is a potential failure point because the arrival of *HaltSender* is uncorrelated to the present status of the sender, and could thus cause *GoA* to go low just as an event ∂req_a arrives, causing a metastability failure on the output of the controlling *send* gate.

Metastability cannot be avoided [Mar81], but there are two methods for increasing the MTBF of the transmitter system in the presence of *halt* conditions. One is to use some form of pipelined [Sei94] or deferred [Mor97] comparison, in which time is traded for failure rate. This will enlarge the size of the receiver buffer by one place for every input request for which action on an arriving halt is deferred (as N_{defh} increases). Another solution is shown in Figure 7.6.

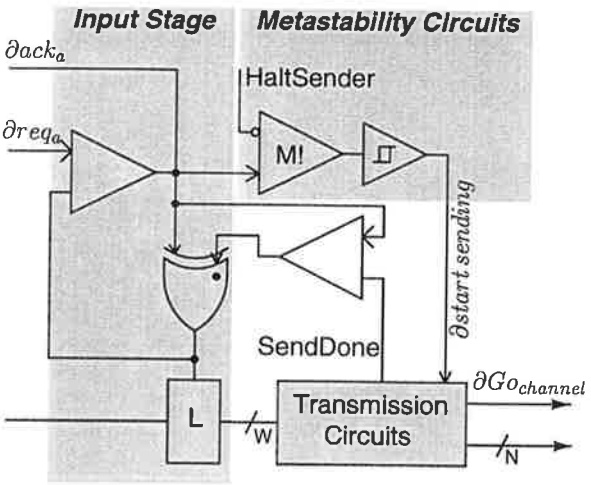


Figure 7.6 NAC Sender Metastability Control.

In this case, the metastability point, the output of the gate marked *M!*, is decoupled from the rest of the sender circuit using a Schmitt-trigger buffer. The output of this buffer will never be indeterminate, although the output node of *M!* can be indeterminate. This only starts the sender when either the *M!* output resolves such that an event occurs, or when *HaltSender* goes low again (when the receiver buffer has sufficiently emptied). This does not *remove* the potential for metastability, but it does prevent control circuits from failing when it does occur. In addition, the time to transmit the packet is bounded unless the time for the receiver to process packets is unbounded, since some time after the generation of the signal *HaltSender*, the receiver buffer will empty and *HaltSender* will go low again, resolving the *M!*

output and starting the sender again. This does cause a throughput degradation because the critical path from ∂req_a to the start of data transmission has two extra gates inserted, which may be problematic if M is low as the extra gates cause a peak throughput degradation, more serious if few packets are sent per *pass* through the input control circuit.

3.1.2 Receiver Control

As the channel width is equal to W, the receiver control is simple. One possible receiver implementation is shown in Figure 7.7.

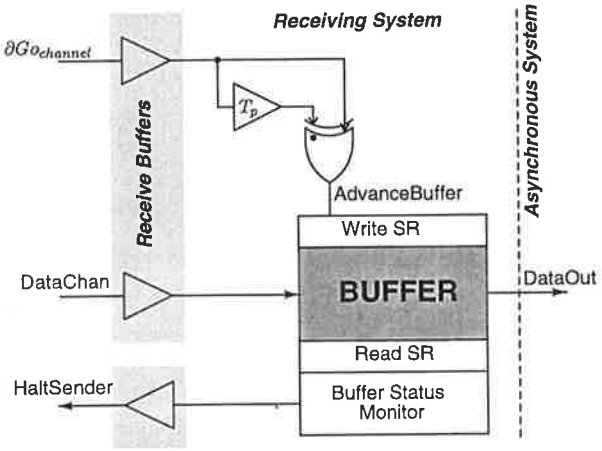


Figure 7.7 NAC Receiver Control for M=1.

The buffer used is a ring-type structure (see page 143) to reduce packet latency in traversing the buffer. A pulse circuit is used to generate *AdvanceBuffer*, which loads the operand on *DataChan* whenever a $\partial Go_{channel}$ event is observed. The process of writing data into the buffer can never overwrite valid data still waiting in the buffer to be used, as the buffer sizing requirement and the *Buffer Status Monitor* (which controls the assertion of a *Halt* used by the sender) ensure that there is *always* space in the buffer to hold any arriving packet.

The output end of the NAC channel, an asynchronous system itself, uses a special interface to the NAC buffer, shown in Figure 7.8.

The buffer indicates that the present output, pointed to by the *Read SR* of Figure 7.7, is valid by asserting the signal *ValidBufferOutput*. This causes the pending event on the input *send* gate to occur, setting *Lt* low and pulsing the *ResetBuffer* signal, which resets the valid status of the entry pointed to by the *Read SR* shift register, and advancing this shift register as well. The output acknowledge, $\partial ack\, out_0$, resets the status of the stage, so that it may receive a new packet from the buffer.

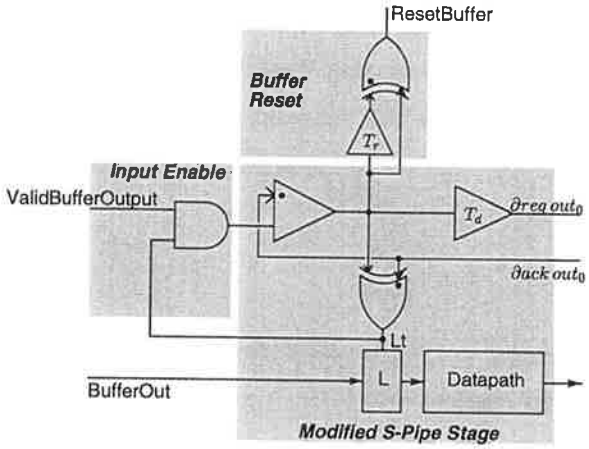


Figure 7.8 Asynchronous Stage for NAC reception.

3.1.3 Buffer Control

The control of the buffers used to hold outstanding operands in NAC requires that the availability of output data be indicated by a logic-level signal *ValidBufferOutput*, used by the asynchronous interface to start computations on the output of the buffer. The buffer control must also indicate when the buffer is nearly *full* so that a halt can be asserted at the sender side. This style of ring buffer control would also be used for *multi-rate* pipelines or any free-flow system requiring a buffer with status monitoring. The interface circuit that indicates output *validity* of the current buffer output word is shown in Figure 7.9.

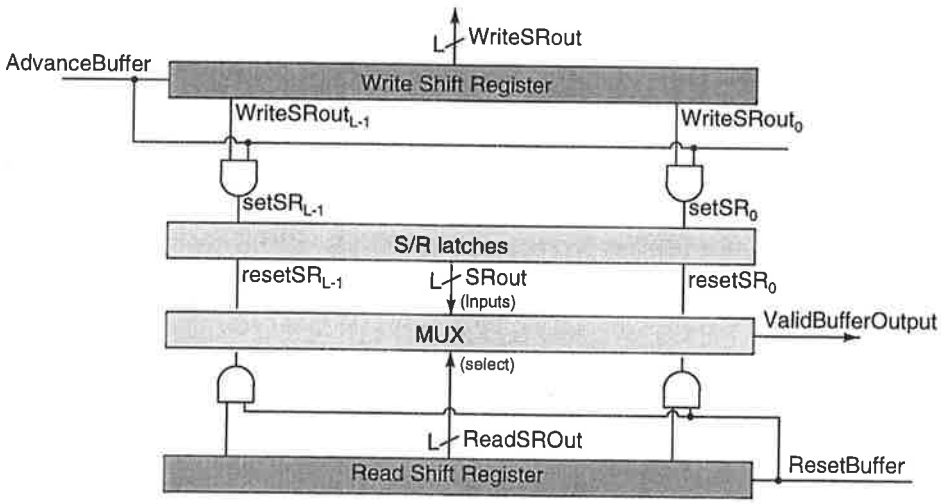


Figure 7.9 Buffer Status Control.

The L S/R-latches of Figure 7.9 indicate that their respective buffer entry is *valid*, and a

multiplexer controlled by the status of the *Read Shift Register* drives one of the S/R-latch outputs to *ValidBufferOutput* to indicate the validity of the buffer entry pointed to by this shift register. When the output stage accepts the relevant valid buffer entry, the *ResetBuffer* line is pulsed, reseting the state of the respective S/R-latch output and advancing the state of the *Read Shift Register*.

The other requirement on the buffer control is that it indicates the buffer *status* — how many words the buffer contains. This is used to set and reset the *Halt* signal sent to the sender when the buffer has the potential to overflow if the *halt* is not sent. This circuit is shown in Figure 7.10.

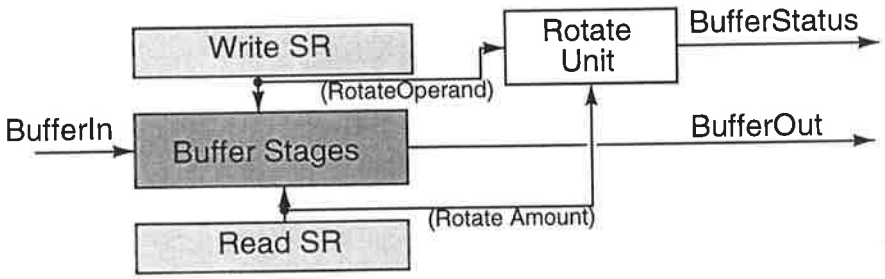


Figure 7.10 Buffer Status Monitoring.

The *Write SR* (indicating which buffer entry is to be written next) and the *Read SR* (indicating which buffer entry is currently driven to the output) are both *one-hot* shift-registers. Therefore, to obtain the status of the buffer, the output of the *Write SR* is simply *rotated* (shifted with wrap-around) by an amount indicated by the *Read SR*. The resultant *BufferStatus* value can be used to control the *Halt* signal, since it outputs a one-hot vector indicating the number of stages currently *full* in the buffer.

3.2 Larger M NAC channels

The wide-channel architecture just described is likely to waste potential data-rate if the input rate is relatively low compared to the available channel bandwidth. This may be a concern in systems with considerable wiring–throughput tradeoffs, an example being inter-chip communication, in which bandwidth cannot be continually increased by increasing wire count, due to pin and power constraints in the transmitter. In such circumstances, an alternative approach can be adopted in which M (the ratio of packet width to channel width) is increased to reduce resource use and improve utilisation of the channel.

3.2.1 Sender Control

The sender control necessary to interface the asynchronous sender to the NAC component is shown in Figure 7.11. The input circuit is identical to that in Figure 7.5, and a high-speed pulse control circuit is required to multiplex out data from the W-bit input packet to N-bit packets for injection into the channel. A pulse-type circuit is used because it would be virtually impossible to construct an asynchronous loop circuit which would control the multiplexer to drive out data at the rate supportable by a high-speed channel. If the channel injection spacing time, T_{ci} were relatively high compared to circuit speed, then an alternative control mechanism would most likely be used.

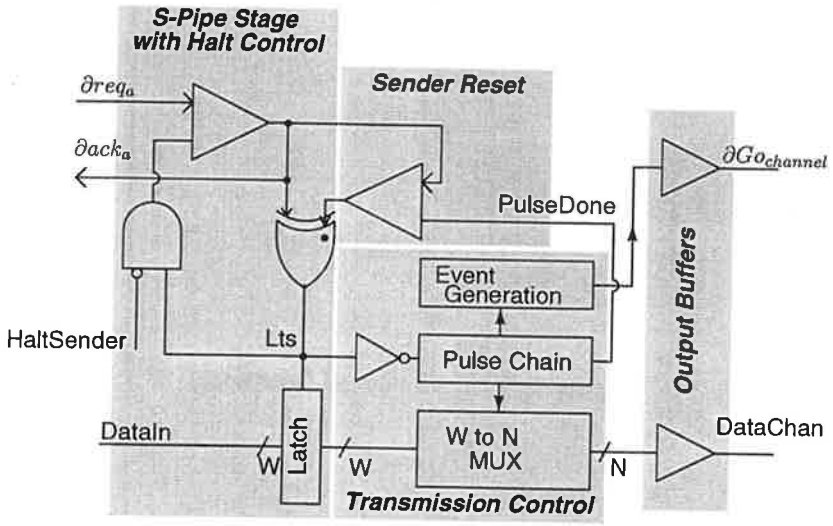


Figure 7.11 General NAC Sender Control.

The pulse chain is a self-timed block that generates pulses that are non-overlapping, thus ensuring that the multiplexer output does not *clash* during pulse switching. The pulse and event waveforms generated by this block for $M = 8$ are shown in Figure 7.12.

3.2.2 Receiver Control

The receiver becomes more complex compared to the simple case considered previously, as multiple requests on $\partial Go_{channel}$ are received before the final W-bit packet has been assembled. A circuit for implementing the receiver is shown in Figure 7.13.

Channel data, arriving on *DataChan*, are loaded into a shift register immediately, and shifted when *AdvanceChannel* pulses. This allows the loading of the first stage of the shift register upon the arrival of data to be concurrent with the generation of the shifting pulse. The *ring* shift register is one-hot, and indicates which part of the shift register has just gone valid

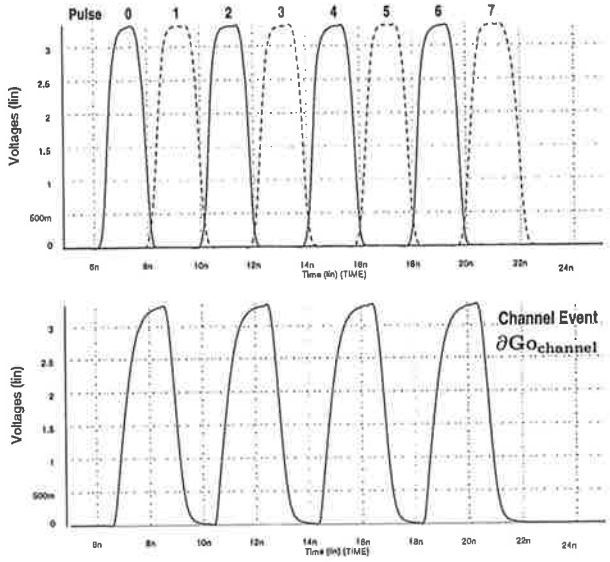


Figure 7.12 Pulse and Event Waveforms in NAC Sender Control. The pulse waveforms, generated by HSPICE simulation, are matched to a 2ns channel injection spacer time. The circuit implementation uses 0.5μm CMOS.

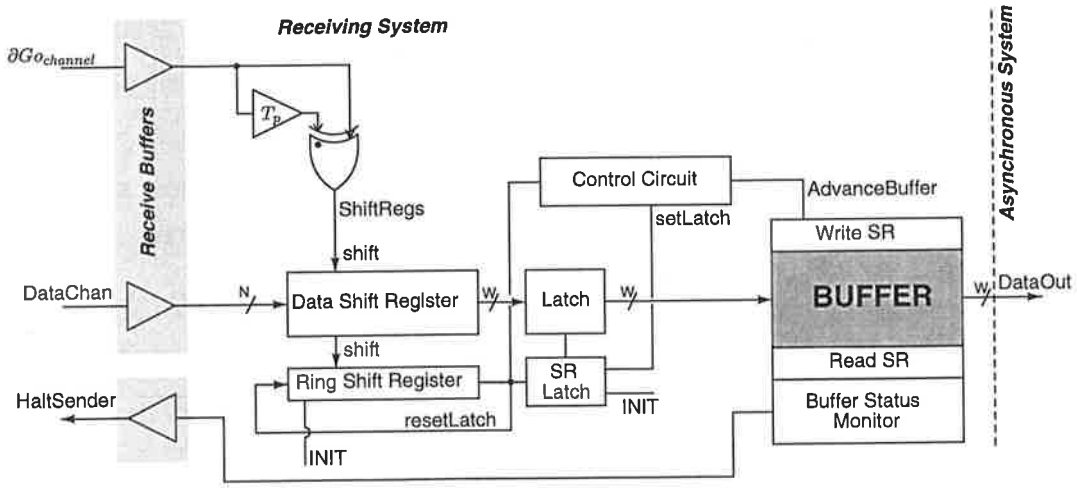


Figure 7.13 General NAC Receiver Control.

after an *AdvanceBuffer* pulse. When the ring shift register reaches the appropriate state (indicating that M items have been loaded into the data shift register), *resetLatch* is pulsed. The S/R-latch output, controlling the loading of data into the latch, is initially high, and *resetLatch* sets it low, thus latching data. This latched data can then be loaded into the main *buffer*, and *setLatch* is pulsed when this task is complete. This allows new data to be loaded into the data shift register while this process completes, not constraining channel bandwidth.

3.3 Performance

The performance of three channel signalling schemes was explored using a chip-level signalling model in a 0.5μm CMOS process.

3.3.1 Channel and Wire Model

The chip-level channel wire model is shown in Figure 7.14. The channel uses a 10mm wire with repeaters inserted at 1mm intervals to effect channel pipelining. HSPICE simulations of this wire model showed an available channel bandwidth of approximately 650MHz, with a latency of 5ns, in 0.5μm CMOS.

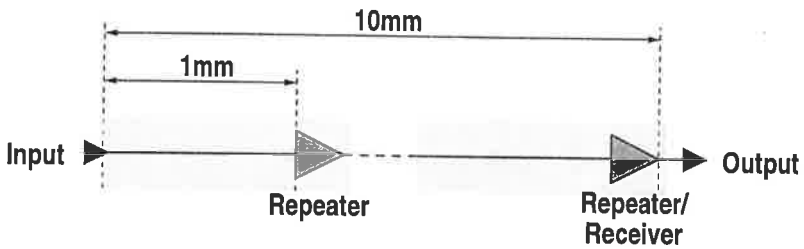


Figure 7.14 On-Chip Wire Model.

3.3.2 System Performance

The performance of three of the schemes for channel signalling was tested using HSPICE simulation of the sender and receiver parts of the circuit, using the channel of Figure 7.14.

The full buffer control of Figures 7.9, 7.10, and 7.13 was not implemented for these tests, since the function of the buffer is to absorb timing variation, not required in such an evaluation. Therefore, the buffer for both NAC system tests is only one place deep, and the asynchronous system at the output is always *free* (ready to receive data).

The asynchronous signalling scheme is two S-Pipes, communicating over the 10mm wire model, and thus is a $M = 1$ fully asynchronous scheme. The wide-channel NAC scheme (in which $N = W$) has the highest potential throughput, however, this is likely to be wasted because the system generating data for the sender is unlikely to be able to do anything

Type	Wires	Utilisation(%)	Packet Throughput	Latency	Merit
Async M=1	10	13%	86 MP/s	6ns	9
NAC M=1	10	61%	408 MP/s	8ns	41
NAC N=1	3	83%	69 MP/s	20.5ns	23

Table 7.2 Signalling System Performance. Using *W=8*, the performance of three communication schemes was explored for a pipelined on-chip CMOS channel. Throughput is given in packet throughput (number of *W*-bit packets per second). The Merit figure is packet throughput/wires. Utilisation is throughput divided by available channel bandwidth.

meaningful at a data rate of 408 million packets per second, and the receiving system is unlikely to be able to process packets at this rate (just the S-Pipe control mechanism, as a FIFO, has a peak throughput of 752 MP/s in 0.5μm CMOS, without any processing and a latch width of 32 bits). Thus, both the utilisation and effective throughput of this channel scheme are likely to be significantly lower in practice.

The one-bit data channel, *NAC N=1*, achieves relatively good throughput considering its minimal resource use. This is because the rate at which the components of the input packet are injected into the channel matches very closely the available channel bandwidth.

4 Conclusion

The architecture and design of free-flow signalling systems have been presented in this chapter. NAC (*Non-Acknowledging Asynchronous*) systems were the forerunner of the free-flow methodology presented in this thesis, and exploit a small subset of the features of the free-flow method. However, NAC is useful in itself as it allows the designer to consider tradeoffs between resource (wire) usage and peak throughput during implementation. The two NAC schemes tested with a chip-level pipelined channel used the two possible extremes of channel width. The full-width channel has the greatest throughput, however this throughput is likely to be wasted in practice because the transmitting system will be unable to generate data at the rates required. Using a single-bit data channel gives greater utilisation at the expense of higher latency and lower throughput. A practical on-chip asynchronous signalling scheme might employ a tradeoff between the two, perhaps at $M = 2$. This would reduce resource usage by approximately 50% with only a minimal impact on latency and usable throughput.

One possible method for the construction and control of the *ring buffer*, used widely in NAC and also in multi-rate free-flow pipelines, has also been described.

4.1 Related Work

STARI [Gre93] is a communication mechanism discussed in Chapter 2.2. It uses a similar method of transmission without acknowledge. This is possible in STARI because both the sender and receiver are *synchronous* systems which operate at the same clock rate, and thus generate and consume data at the same rate. STARI is designed to decouple skew considerations from the implementation of the channel between two such communicating synchronous systems. STARI also uses an encoding scheme for the channel wires that packs timing information together with the data.

NAC is a fundamentally different scheme from STARI, as the sender and receiver are *asynchronous*. This implies that the operation rates of the two systems may vary considerably, and thus some form of communication between the two systems is required so that data is not lost in the transmission process. The use of the FeFA approach allows the bandwidth limitation that this would normally impose to be eliminated, and the use of a *buffer* as part of the receiver allows a number of packets to be *in-flight* between the sender and receiver concurrently.

Yantchev [YHJN95] proposed an asynchronous protocol that sends parts of the total word without acknowledegment, only requiring acknowledegment at the end of the reception of the entire word. A modification to this protocol would be to allow the *reception* of the acknowledgement to come behind the start of reception of a new word, removing the bandwidth limitation imposed (as discussed before). It would be interesting to see how such a scheme could be implemented in practice.

Chapter 8

Amedo — A Free-Flow Microprocessor

MICROPROCESSORS provide a vehicle of considerable power for exploring a new design methodology as there are many different, interacting facets in their design. The complexity of microprocessors can also force the design approach to both confront and find solutions to problems encountered in such a process. For these reasons, the application considered in detail for *free-flow* is an advanced microprocessor. This also allows previous work based on the design and analysis of the ECSTAC processor to be built upon, and the issues raised by the design to be fully addressed.

Through an exploration of the characteristics of compiled code, unique architectures are developed that are highly suitable for the asynchronous execution of integer programs. Using the Free-Flow methodology, an asynchronous microprocessor is developed which exploits both circuit timing and dynamic instruction behaviour characteristics. Amedo (**A**synchronous **M**icroprocessor **E**xecuting **D**LX c**O**de) is shown to improve performance across a range of sample benchmarks by up to 81% over an idealised RISC implementation of the chosen architecture.

Many researchers have realised that the performance deficit of pure asynchronous pipelines (compared to synchronous implementations) is difficult or impossible to overcome, and have proposed varying degrees of superscalar architectures to realise performance benefits. Some are relatively simple extensions to traditional superscalar techniques [Ric96, ECFS95], while others have some novel features desgined to make result flow more efficient [MRW96, SSM94]. Two distinct machines are SCALP and ECSCESS. SCALP [End95], designed explicitly for low-power, builds upon *transport-triggered* architectures [Cor93] to avoid dependencies on

a global register bank, instead using a large number of functional units that are referred to *by name* (instead of implicitly being chosen by the hardware dynamically), with each having a result queue that can be accessed by other units. ECSCESS [Mor97, Chapter 8], a successor to ECSTAC, used an architecture similar to SCALP for obtaining higher performance. Despite the efficiency of ECS pipelines, long asynchronous pipelines tend to cancel any real performance benefit from using data-dependent computations, and ECSCESS is a flattened machine which employs a wider degree of parallelism with little or no unit pipelining, allowing performance gains from data-dependent computations to be realised in a superscalar machine. The author reports a simulated peak speed of 85MIPS in the presence of branches, using randomised code and parameters equivalent to a 0.8μm CMOS process.

Recent work, spurred by the increasing use of multiple-instruction issue-per-cycle architectures [Joh91], has shown that the available levels of ILP in typical programs are severly limited [Wal90, JW89]. Current designs are tending to move towards extremely high operation rates with balanced degrees of parallelism [GBD+96]. Thus, asynchronous designs cannot rely solely upon parallelism to provide a realistic alternative. Therefore, the primary goal of this chapter shall be to develop a highly-efficient *pipelined* asynchronous implementation of an architecture designed for clean pipelining and low cycle time, using an asynchronous methodology geared for highly-efficient pipelining. The desciption shall cover mostly the *architecture*, with descriptions of the *circuits* used only where critical issues arise.

1 Base Architecture

The initial stages of the architecture exploration for the Amedo processor focused on characterising the performance of a simple pipelined RISC machine. The instruction set of this *base* machine is DLX [PH96], and is similar to the commercial MIPS architecture [KH92]. A RISC approach is employed to keep control complexity down, so that the machine can focus on efficient pipelining rather than complex control tasks. DLX is a well-documented machine [SK96a] for which compilers are available, and has a simple load/store RISC orthogonal instruction set (summarised in Appendix B). The DLX ISA contains no explicit support for features not necessary in a prototype machine — virtual memory, system support, and detailed, specific routines and procedures for dealing with interrupts and exceptions.

1.1 DLX Instruction Set Overview

DLX has just three basic types of instructions, the formats for which are shown in Figure 8.1. J-type instructions contain one 26-bit field used to form an offset relative to the current program address for branches. I-type instructions use one source and one destination register,

contain one 16-bit immediate field, and are used for immediate integer arithmetic instructions, conditional control transfers and register-based jumps, and memory load/store instructions. R-type instructions use two source registers and one destination register, contain a 11-bit field used to specify the operation type, and are used for specifying integer and floating-point operations.

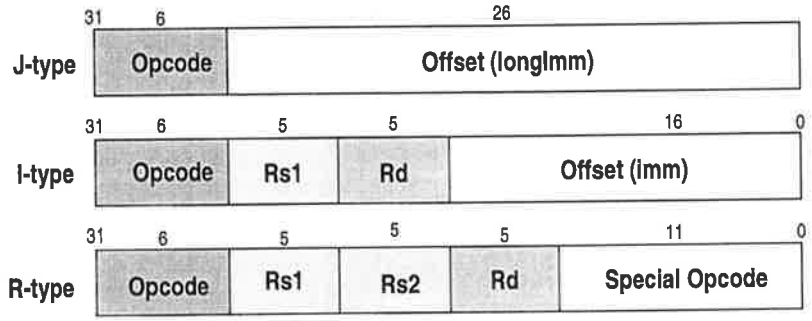


Figure 8.1 DLX Instruction Formats. The numbers shown indicate the width of the relevant field in the instruction.

Modifications to DLX ISA

Three modifications are made to the standard DLX instruction set for the Amedo ISA. A NOR register-register instruction is provided to enable simple ones-complementing of an integer register. The operation of the integer multiply and divide instructions is moved to the integer unit. Unlike DLX, integer multiply/divide instructions are executed directly on integer registers. In addition, offsets in control transfer operations refer to *word* quantities, the size of instructions. This has the effect of allowing all control transfers to go four times as far.

1.2 Base Machine

The base machine used for exploration of performance and instruction execution characteristics is shown in Figure 8.2, and is *synchronous*. This synchronous machine model is used to perform preliminary performance estimations and to assess the impact of some architectural improvements to be considered.

The relevant base architectural features are

- EX and MEM result forwarding
- 1-cycle load interlock delay
- 1-cycle mispredicted branch penalty
- *perfect* memory hierarchy

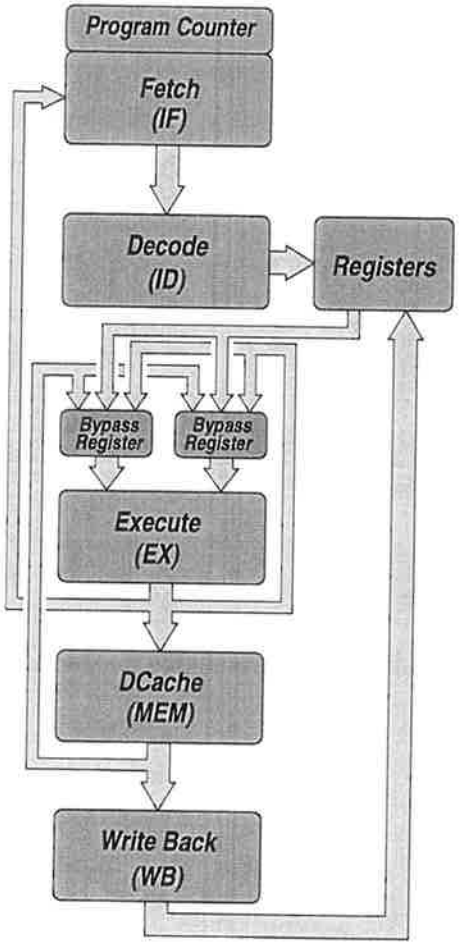


Figure 8.2 Base Architecture. The basic architecture is a classic five-stage RISC pipeline with result bypassing, a one-cycle load interlock delay and a one-cycle mispredicted branch penalty.

- near-zero-time system calls

A load interlock delay is required when an instruction immediately succeeding a load instruction requires its result, resulting in a *load interlock*. The branch penalty is taken to be one cycle due to the time taken in the EX stage to both determine that a control transfer is taken and to perform the addition to determine the target address and subsequently update the program counter, by which time the *Fetch* unit will be deep into its access cycle, unable to abort, change its fetch address and still supply the instruction by the next clock cycle.

To simplify the initial model of the machine, a perfect memory system is assumed, that is, all memory accesses take exactly one clock cycle and never stall. System calls to functions provided by the operating system take zero time once the appropriate TRAP instruction is issued, and every instruction takes one cycle (even integer multiply and divide) in the execute stage. This is acceptable for the current model as integer multiply and divide constitute a very small portion of the dynamic instruction count (at most 2%), and floating-point performance

will not be considered in this study.

This base machine is almost identical to early RISC machines that employed linear pipelines with little or no functional unit parallelism [KH92, HB90]. The assumption of an idealised memory system simply removes one source of complication from the model, since the memory system would be heavily implementation dependent.

1.2.1 Base Machine Cycle Time

The cycle time of a simple RISC machine is usually set by the time taken for a full word-width addition in the EX stage of the machine [Cho89]. The critical path in the EX stage for a synchronous DLX machine is shown in Figure 8.3.

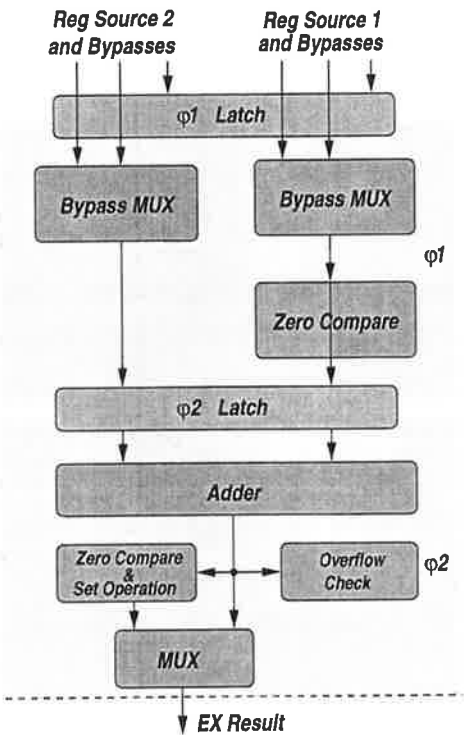


Figure 8.3 Critical Path in EX stage of Base Machine.

In 0.8μm CMOS, this addition using a precharged MCA-style adder [WE95] with two levels of carry bypass takes approximately 5.0ns, including sum logic time, from HSPICE simulation of extracted layout. The zero detection and set operations take approximately 1.2 and 1.0 gate delays, respectively, and the output MUX takes 1.0 gate delays. Therefore, the base

machine cycle time is estimated as

$$\begin{aligned} T_{\phi 2} = T_{cycle}/2 &= T_{latch} + T_{add} + T_{zero} + T_{set} + T_{MUX} \\ &= 1.0ns + 4.0ns + 1.0ns + 0.8ns + 0.8ns \\ &= 7.6ns \end{aligned}$$

thus giving 66MHz operation, assuming that the generation of the propagate and generate signals for the adder are pushed into $\phi 1$ (thus saving about 1.0ns from the latency of the critical path), and that a dynamic latch is used. This assumes the base machine uses *Single Phase Clocking* [DAC+92], with two latches per pipeline stage and a low-skew clocking scheme.

1.3 Performance Modelling

The base machine is used as a comparison point against the asynchronous machine model which employs more advanced architectural features to improve performance. The merits of the asynchronous machine can be examined relative to this base machine, despite its idealised features. The asynchronous machine will employ a similar pipelined architecture with no functional unit parallelism. The architecture to be considered improves the performance of the pipelined machine using free-flow, and would be impossible to apply to the synchronous machine.

In order to model the performance of the microprocessor, a full cycle simulation model was developed, akin to RTL [Bre75] written in C that executes binaries on a model of the machine. An existing compiler [S+95] was used to generate DLX assembler, and a custom assembler was used to convert this representation to the required binary format before being executed on the specially-designed simulator. This simulator *exactly* models the pipelined behaviour of the system, load stalls and branch delays, and result forwarding, as well as providing a large number of *hooks* so that the dynamic behaviour of the program can be analysed. This simulator was used to generate all the experimental data to be described.

1.3.1 Benchmarks

Relatively small integer benchmark programs are used to determine the relative performance of the architecture to be discussed. Although such an approach is fraught with danger, there are good reasons to use small programs to determine performance improvements.

In this approach, precise performance comparisons against potentially competing architectures are not desired, nor are the optimum caching and branch prediction strategies across a wide range of applications. General performance trends across very similar architectures that execute identical code are being sought, and the results are not expected to be exact. These

programs, not containing any special optimisations on individual architectures, will provide a good indication of what compiled code will look like. Thus, the performance measurements, if positive, will apply in general to compiled code programs.

In addition, the simulator, because of the modelling of pipeline structure and detailed statistical updating, cannot execute large programs in reasonable time. Furthermore, these larger programs require access to a large number of system functions not provided by the simulator. Even "integer" benchmark programs employ some degree of floating-point execution (for example, SPEC95 [Cor97]), for which a realistic model of execution has not yet been developed for Amedo. Therefore, it was decided to model integer performance only for this preliminary evaluation. The benchmarks employed are described in Table 8.1.

Name	Description
Dhrystone (V2.1)	Synthetic benchmark designed to mirror properties of typical integer code
Dhampstone	Modification to Dhrystone to model system call functions
Hanoi	"Towers of Hanoi" solver
NSieve	"Sieve of Eratosthenes" for generating prime numbers
Queens	Determining legal placement of queens on chessboards
Heapsort	Sorts a large array of randomised integers
Fib	Fibonacci number generation

Table 8.1 Preliminary Benchmark Programs. These simple integer benchmarks were the first set to be evaluated on the base architecture.

The use of a limited set of benchmarks to predict the performance of general applications is known to be potentially misleading [Wei97]. The use of a limited range of small integer benchmark programs will restrict the general validity of any absolute performance figures derived, instead only giving rough measures of performance gains. Again, absolute performance metrics for this architecture are not sought, and small benchmarks are valid for a limited preliminary evaluation of the Amedo architecture. Before embarking on a real implementation of the processor, its performance on a wider range of benchmarks would need to be evaluated to enable a more complete set of design decisions to be made, especially any concerning cache sizing and branch prediction.

1.4 Benchmark Performance and Statistics

The code simulator was enhanced with a large instrumentation package that allowed the behaviour of various interesting parts of the machine to be analysed. This was used to perform

extensive measurements on the dynamic behaviour of the base machine. All benchmarks were compiled to DLX assembler using `gcc -O3`, producing the best execution times on the base machine. The *instruction mix* for each of the benchmark programs is shown in Figure 8.4. This dynamic trace of the benchmarks shows that there is a wide range of behaviours in these

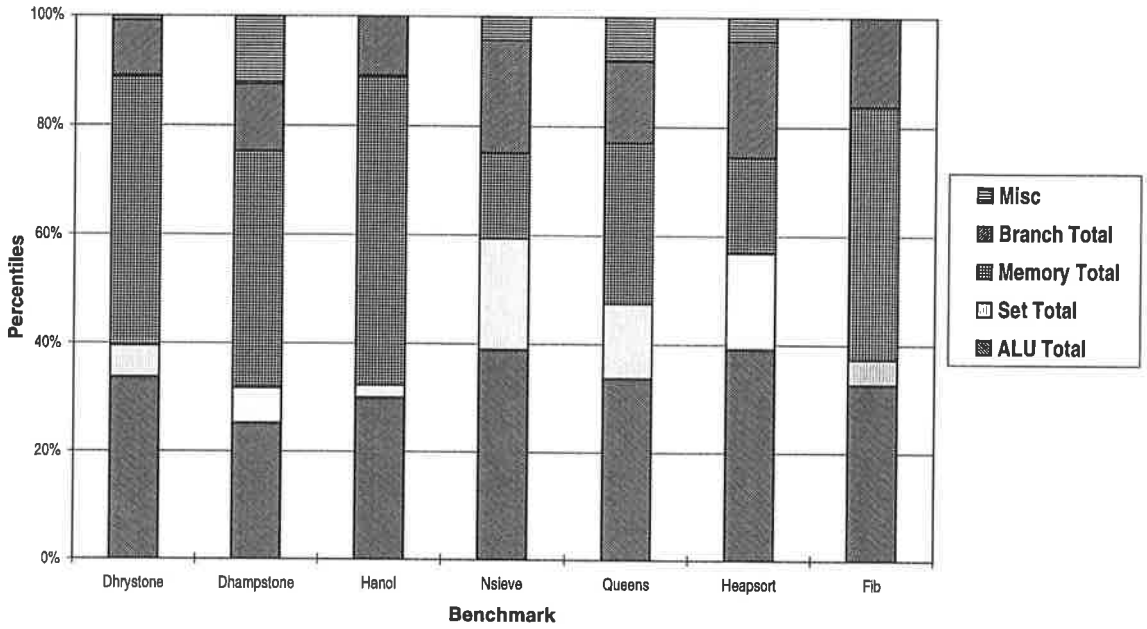


Figure 8.4 Instruction Mix.

programs — they are not uniform or totally specialised. This is good from the perspective of performance evaluation using these codes, since any architecture that show a net gain across these applications is more likely to apply in general, although not of the same values.

The number of cycles lost to taken control transfers is shown in Figure 8.5. Across all benchmarks, the branch loss accounts for at least an 8% increase in CPI (Cycles-Per-Instruction) on the base machine. Obviously, to mitigate this source of performance loss, some form of branch prediction will have to be employed. However, the detailed behaviour of branches is not considered here as branch prediction is a relatively well-understood design problem [PH96, TV95].

The size of a *basic block*, the amount of code between a control transfer operation (taken or not), is shown in Figure 8.6, and shows how many instructions can be expected between adjacent control transfer operations. The range of behaviours shows that, although some benchmarks execute relatively many instructions between branches (up to 10), many execute relatively few (around 4).

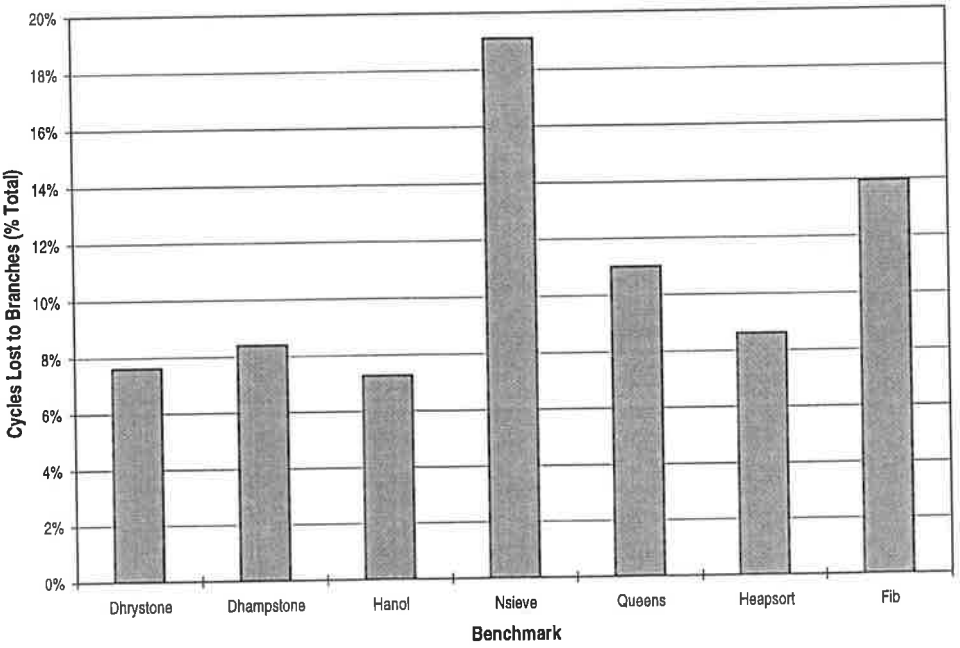


Figure 8.5 Branch Cycle Loss. No prediction is done on any control transfer, resulting in a high CPI loss in the base machine. Cycles lost are the percentage of total opcodes executed.

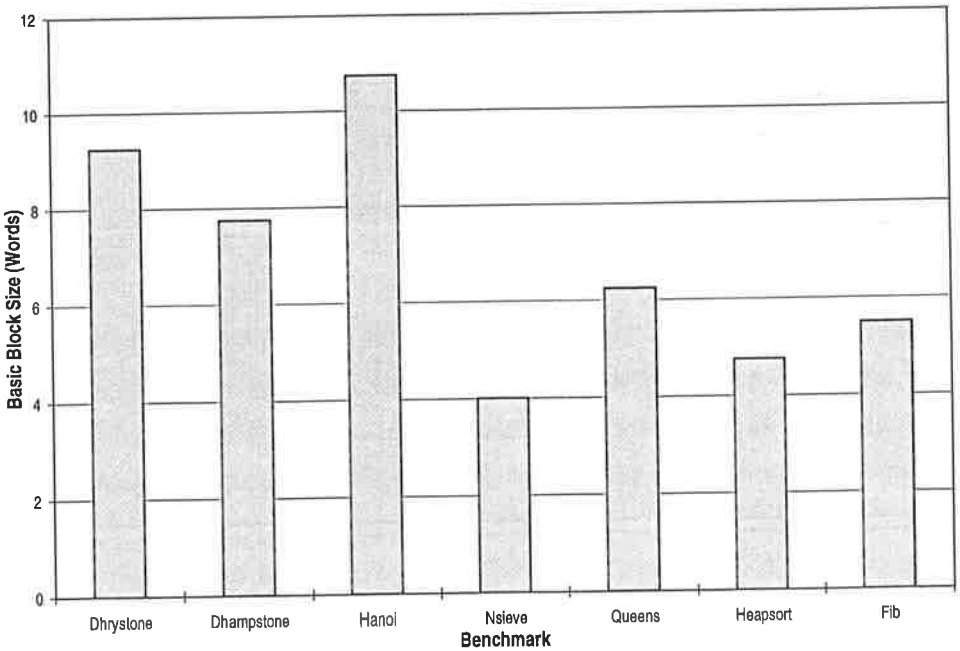


Figure 8.6 Basic Block Sizes. The basic block size is the number of instructions between control transfers (taken or not).

1.4.1 Load Behaviour

The behaviour of memory references, and loads in particular, will be critical in the Amedo processor, as the data cache is a large unit that is on the critical path of the processor. Methods for improving memory access times are sought for the Amedo machine to obviate the bottleneck posed by the memory stage. The relative frequency of load stalls, instructions with source operands which depend on a memory instruction which issued on the previous cycle, is shown in Figure 8.7.

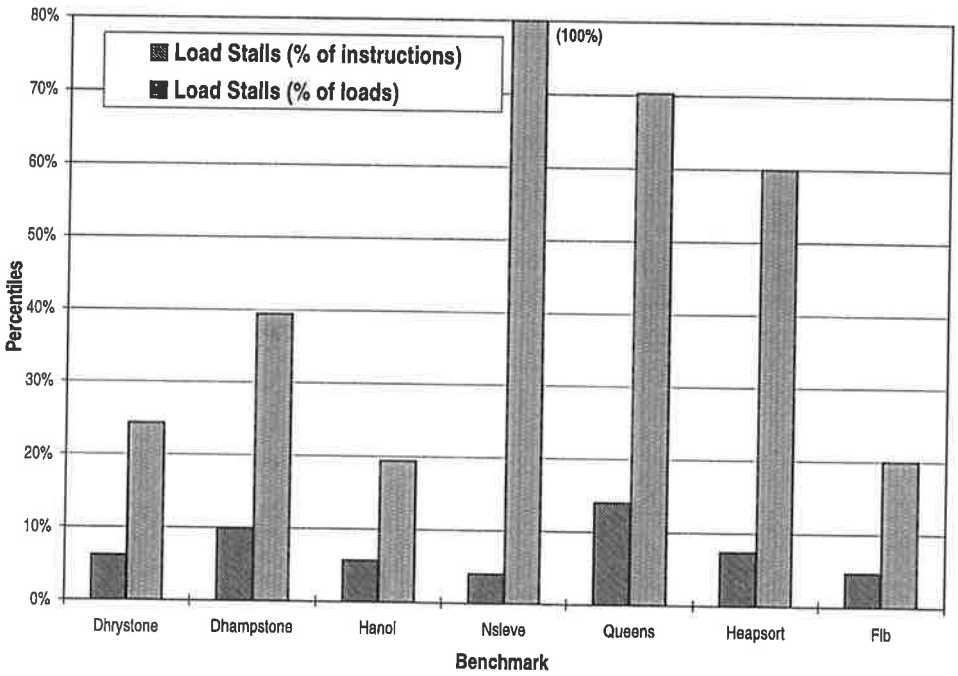


Figure 8.7 Load Stalls in the Base Machine.

Load stalls represent another source of CPI loss in the base machine, although not as dramatic as the cycles lost to taken branches (Figure 8.5), and cause a CPI increase of between 4% and 15% per instruction. Past work on data caching [Joh91, Bra93, Dea92] has attempted to improve the cycle time of the process by using specialised caches that can improve total machine cycle time, since the DCache access in the memory stage is typically a component of the critical path of the machine. However, these approaches do not eliminate load stalls because an instruction issuing after a load and dependent upon it must still stall while this result is produced in the memory stage of the pipeline, even though the complexity of the critical path in the memory stage may be reduced. A method is sought to reduce the number of load stalls occuring, similar to the objective of branch prediction.

The simulator was used to trace the *source registers* used for a load, for example

```
lw      r7,r1,#-4    ; r7 is load destination, r1 is source, #-4 is offset
```

The results of this experiment are shown in Figure 8.8. This shows that the compiler is *tending* to use a relatively small subset of index registers over the total run of the benchmarks, although this is generally more pronounced on benchmarks which have heavier loads of memory accesses. In any case, this is to be somewhat expected since the compiler must retain a significant portion of the register space for variables, temporary storage, and system functions (like stack, frame and return address pointers).

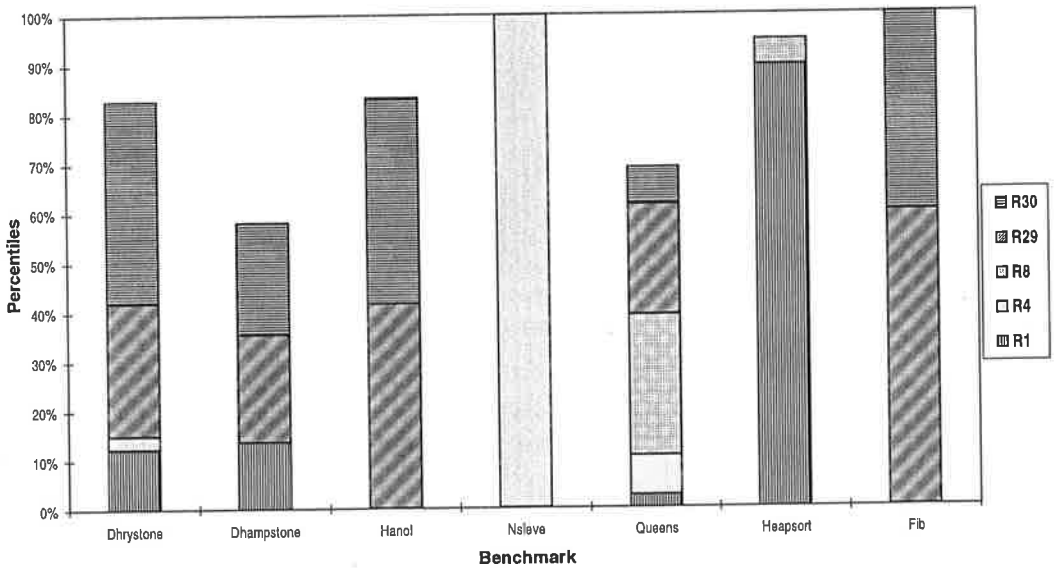


Figure 8.8 Source Registers for Loads. The compiler has a tendency to use only a small subset of registers as index registers for memory loads.

Load Stall Elimination and Data Caching

Instrumentation was added to determine how many load source registers were reused without being modified. An example of code performing in this way would be

```
lhi     r1,(data_address >> 16)        ; load high 16 bits of r1 with addr.
addui   r1,r1,(data_address & 0xffff)  ; setup low 16 bits of address
lw      r4,r1,# 0                      ;    in r1 for load, and load r4
lw      r5,r1,# 4
lw      r7,r1,# 8                      ; load values
add     r8,r5,r7                       ; add values loaded, r7 causes load stall
```

In this example, once the address index register is loaded, `r1` in this case, other registers are loaded using offsets from `r1` without the value of `r1` being modified. The third load instruction also causes a load stall because its result, `r7`, is required by a subsequent instruction.

Figure 8.9 shows the frequency of these kind of accesses, and the behaviour of the offset values used to index load values.

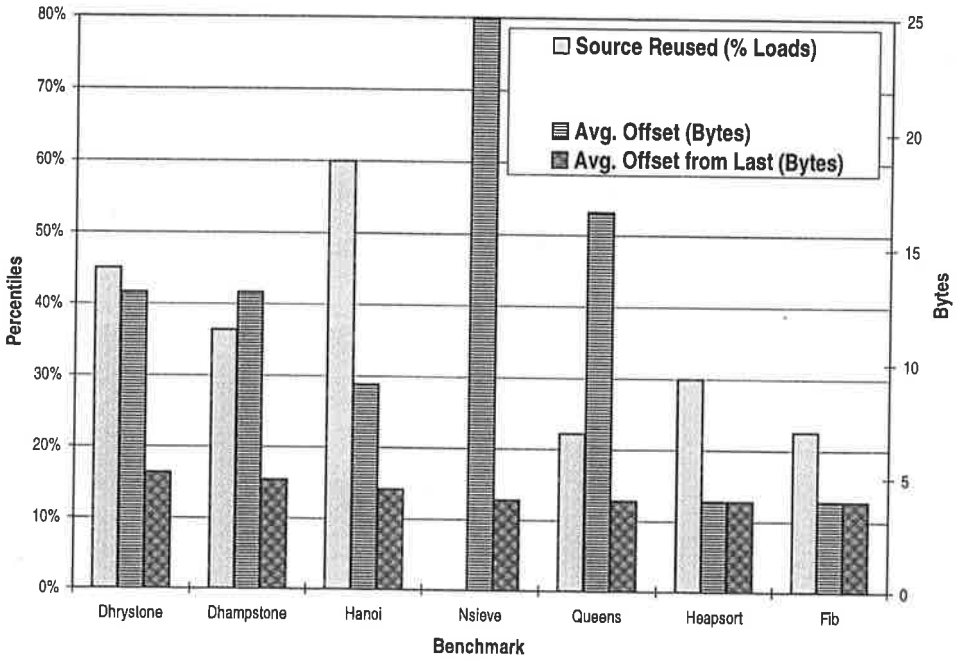


Figure 8.9 Register Reuse for Loads. The *Average Offset* is the average offset used in a source register for a load which is not modified, and *Average Offset from Last* indicates how much the offset changes from one load to the next which re-uses the same source index register for the load.

Even this simple observation shows some promising characteristics. From 10% to 60% of the sources are reused without modification, and most have relatively local offset characteristics. In Section 2, the potential performance of load-stall eliminating caches and the structures required to implement these special caches in the Amedo machine will be explored in detail.

1.4.2 Temporal Behaviour of Loads and Stores

The dynamic instruction behaviour immediately following loads and stores was investigated. This will become important if *dead-time* in the memory stage (when non-memory-referencing instructions are passing through) is utilised to perform pending tasks or to perform prefetching into the DCache memory array. The temporal behaviour of loads is shown in Figure 8.10.

Some ways in which these dynamic characteristics can be used will be explored in the architecture design of Amedo. Utilising the temporal behaviour of stores may also be useful. Stores produce no result for the Writeback stage of the pipeline, and if stores tend to be followed often by ALU operations, or even loads, then the store can effectively be *pushed* into the MEM stage and not hold up subsequent instructions (since the store does not produce a result). The cycle time of the memory stage will only increase for a store when the

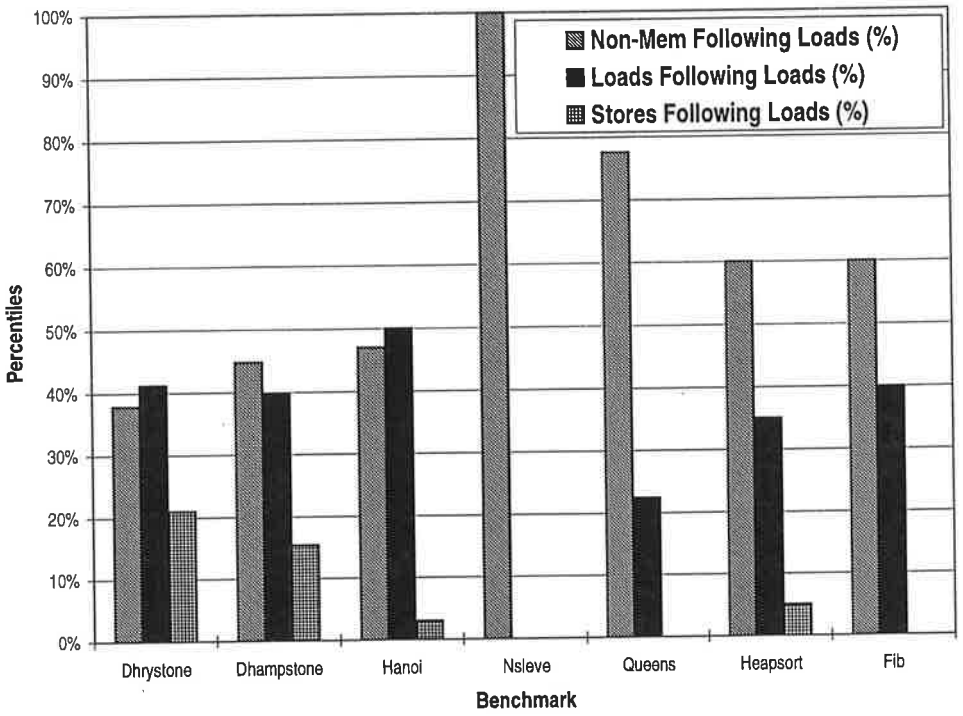


Figure 8.10 Temporal Load Behaviour. This shows the types of instructions following loads as a percentage of all instructions following loads.

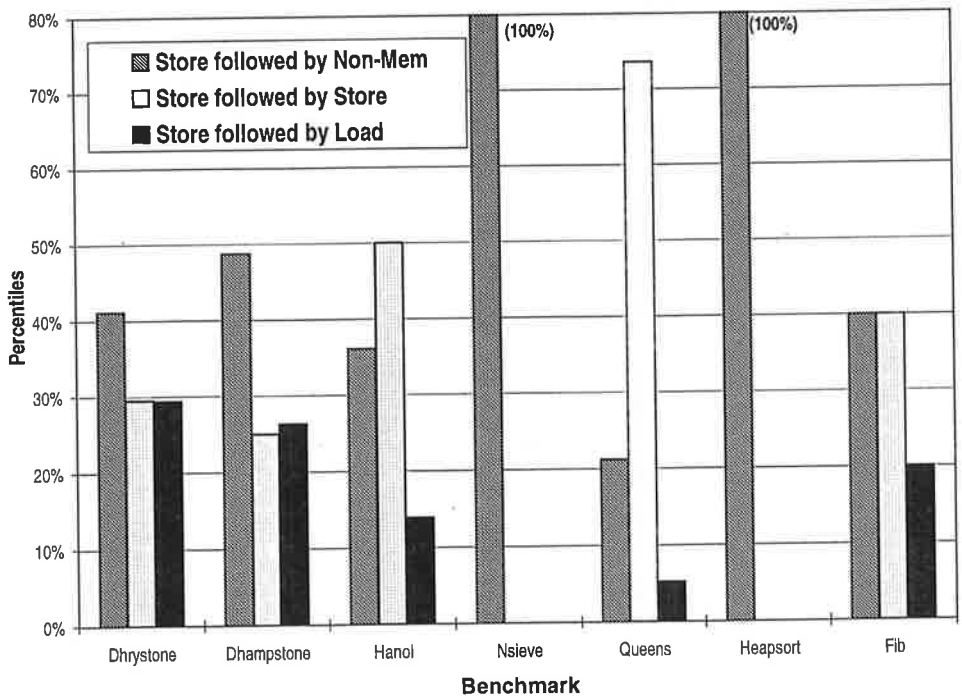


Figure 8.11 Temporal Store Behaviour. This shows the types of instructions following stores as a percentage of total instructions following stores.

instruction that preceded it was itself a store, meaning that the store must be fully processed. The behavioral characteristics of stores are shown in Figure 8.11. In most benchmarks used, the majority of stores are followed by non-memory operations.

2 Amedo Architecture

The architecture of Amedo is shown in Figure 8.12. The processor is split up into autonomous units which perform major required tasks that are separate from other units. The *N-Pipe* fetches instructions for execution, performs branch prediction and program counter control, and interacts with the *N-Box*, which contains the memory array for instructions (as in EC-STAC, a split cache architecture is employed, as opposed to a unified cache). The *X-Pipe* is the execution core of the processor, containing the *ID-EX-MEM-WB* stages of a classic RISC machine. The *MEM* stage of the X-Pipe interacts with the *M-Box*, which contains the memory array for the DCache. Both the N-Box and M-Box are expected to contain their own prefetch and access control logic, and both interface to the E-Box, which controls access to the external memory system of the machine.

The N-Pipe and X-Pipe, the critical execution core components of Amedo, shall be the primary concern of this chapter. The N-Box and M-Box are abstractions of the cache units, and are thus not particularly important in an exploration of this kind except for an understanding of their access timing and characteristics. The E-Box, an abstraction of the external interface, would be an exercise to be undertaken when the machine is implemented, and is not relevant for this analysis.

The N-Pipe and the X-Pipe are *decoupled*. The fetching of instructions and the prediction of branch targets is not the concern of the X-Pipe's execution of the information contained *within* the instruction, although some interaction does occur when branch outcomes are known. The N-Pipe and X-Pipe thus constitute a multi-rate free-flow system. Instruction Decode is the point where timing factors for the X-Pipe relating to instruction execution are determined.

2.1 X-Pipe

The internal structure of the X-Pipe is shown in Figure 8.13. There are four stages, corresponding to the four execution stages of a traditional RISC pipeline. The Decode stage, ID, is the interface to the decoupling buffer between the N-Pipe and the X-Pipe, and is the *Issuing* stage for the X-Pipe. The Execute stage, EX, Memory stage, MEM, and Writeback stage, WB, all correspond to typical RISC stages. The *forwarding* path from the EX and MEM outputs is also shown — forwarding control will be described in Section 2.1.4.

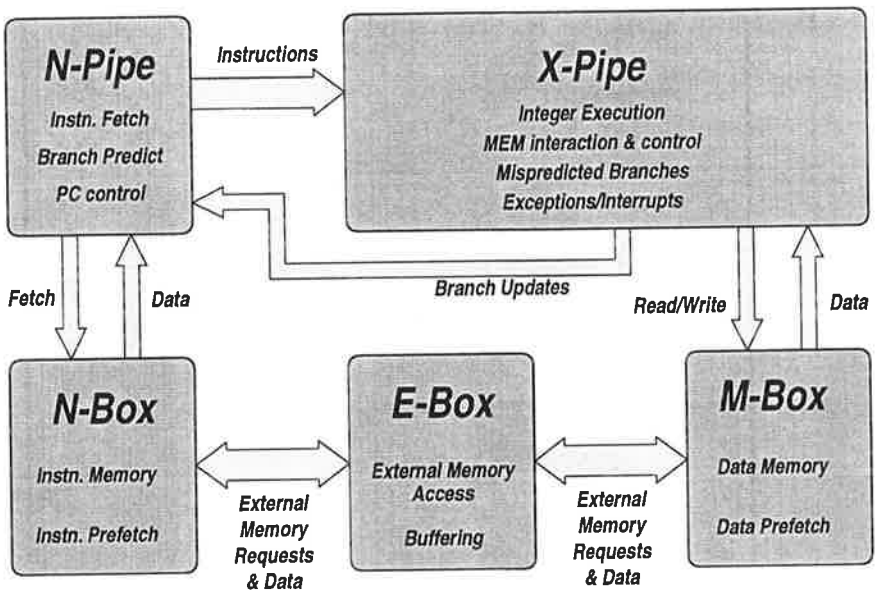


Figure 8.12 Amedo Structure.

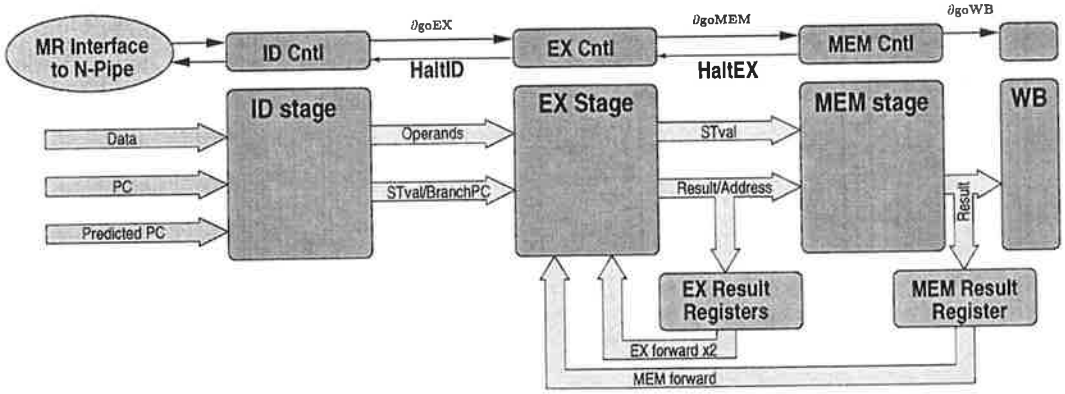


Figure 8.13 X-Pipe Internal Structure. The X-Pipe is short, only four stages long, resulting in a minimal cycle time loss due to uncancelable delay skew.

In subsequent unit designs, it is *expected* that the N-Pipe is doing some form of branch prediction. Hardware in the X-Pipe is designed to determine correct outcomes, updating when a misprediction occurs, and is not constructed for performing branch target calculations.

2.1.1 Instruction Decode Stage

The internal structure of the Instruction Decode stage is shown in Figure 8.14. This only shows the components necessary for setting up instruction execution for the various operand types.

Registers are accessed on *every* instruction. Although J-types do not require register access, they are infrequent and selecting register accesses based on instruction type adds overhead

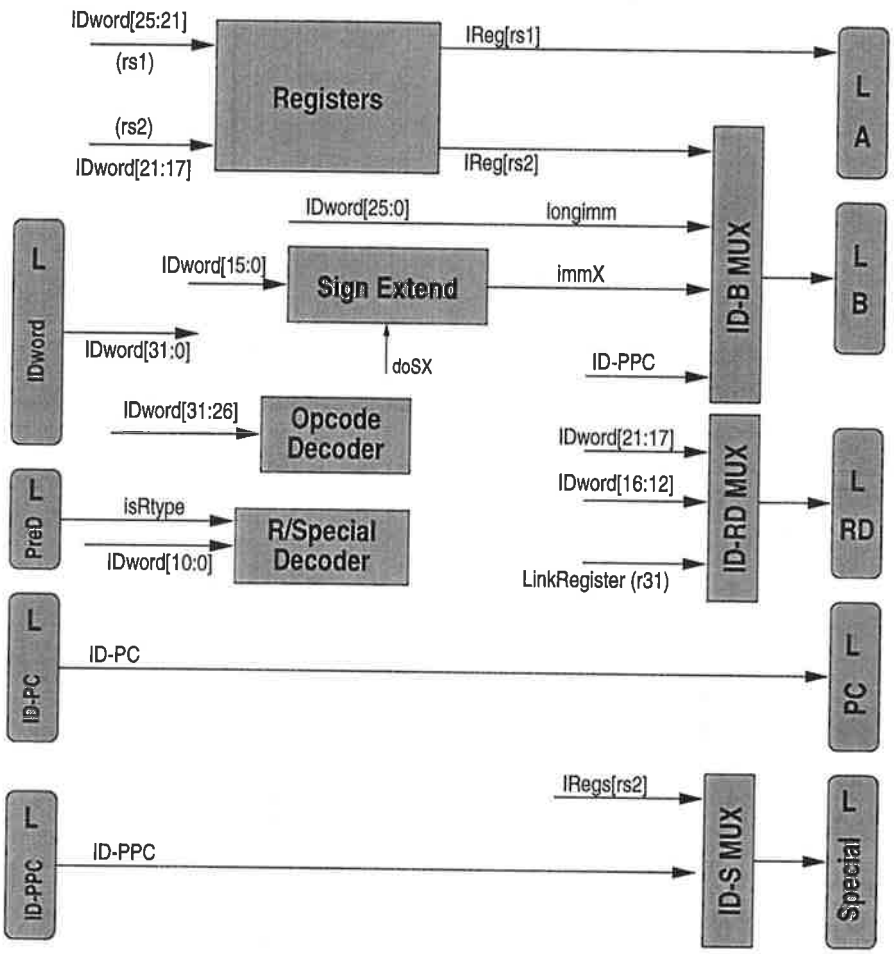


Figure 8.14 Amedo ID stage structure.

to the critical path. Apart from J-types, every instruction has at least one source register, and this goes directly to *L-A*, the latch for the A operand.

The *ID-B MUX* is used to setup data for the second operand used by EX. Although J-types are predicted perfectly by the N-Pipe, TRAP instructions are not predicted since they will involve heavy control in the execute stage. TRAP instructions are J-type and use the J-type *longImm* field to specify the trap code, and this is required for the *B* operand during trap handling. I-type instructions require the immediate field for operand *B*, which may have to be sign extended (performed when *doSX* is asserted). Branch instructions always have the *branch target* available on *ID-PPC*, the predicted PC from the N-Pipe, regardless of whether the branch is predicted taken or not. For register-based jumps, *L-A* receives the *true* branch target, and is compared with the *predicted* target from the N-Pipe on *ID-PPC*. *ID-PPC* is used for operand *B* for the execution of these instructions.

The destination register depends on the type of instruction. In addition, linked-jump instruc-

tions, JAL and JALR, implicitly set the destination register to the link register, r31. *ID-RD MUX* selects between the three possible alternatives.

The *ID-S MUX* receives data for only two types of instructions. During stores, register data from *Iregs[rs2]* goes to the MUX — on all I-type instructions other than stores, the fetch of *rs2* is useless because it corresponds to the destination register, but for stores *rs2* is used to encode the source data for the store. During a conditional control transfer, the *ID-S MUX* receives the target PC of the branch (from *ID-PPC*), used to update the N-Pipe if the branch is mispredicted and taken.

Critical Path

Predecoding performed in the N-Pipe allows any decoding done in ID to be off the critical path, which is formed by the access of the register file and the subsequent multiplexing of data through *ID-B MUX*. The total access time of the register file was determined by HSPICE simulation of 0.8μm CMOS layout of a 32-bit, 32-entry self-timed register file similar to that used in ECSTAC, with two read ports and one write port. The access time of this structure is 4ns, including address drive time into the array and an output load equivalent to eight gate loads (200fF). The ID critical path is thus

$$\begin{array}{ccccccc} latch & \rightarrow & reg\ access & \rightarrow & MUX & & \\ 1.3ns & + & 4ns & + & 1ns & = & 6.3ns \end{array}$$

Although J-type instructions in DLX do not require register access, they are relatively infrequent and making a special timing case for them would not give any tangible performance benefit (probably the reverse, due to added control complexity). The design of the register array incorporates self-timed precharge, activating as soon as the result has been sensed. This allows the array to precharge before the next access arrives, independently of the stage control.

2.1.2 Execute Stage

The structure of the Execute stage is shown in Figure 8.15. The execute stage receives four 32-bit data values, in addition to control signals for the operation. The operands for the current operation are latched via *L-A* and *L-B*. The store value and taken branch target value are latched via *L-S*. Finally, the program counter value for this instruction is latched via *L-PC*.

The three operand inputs require *bypassing* when the fetched operand depends on one of the previous two results computed in the X-Pipe. This is handled by the Bypass MUXs on the input for each operand, and control signals for these forwarding multiplexers are setup in the ID stage and latched by EX. The operation of the EX stage is then dependent on the type of instruction being processed.

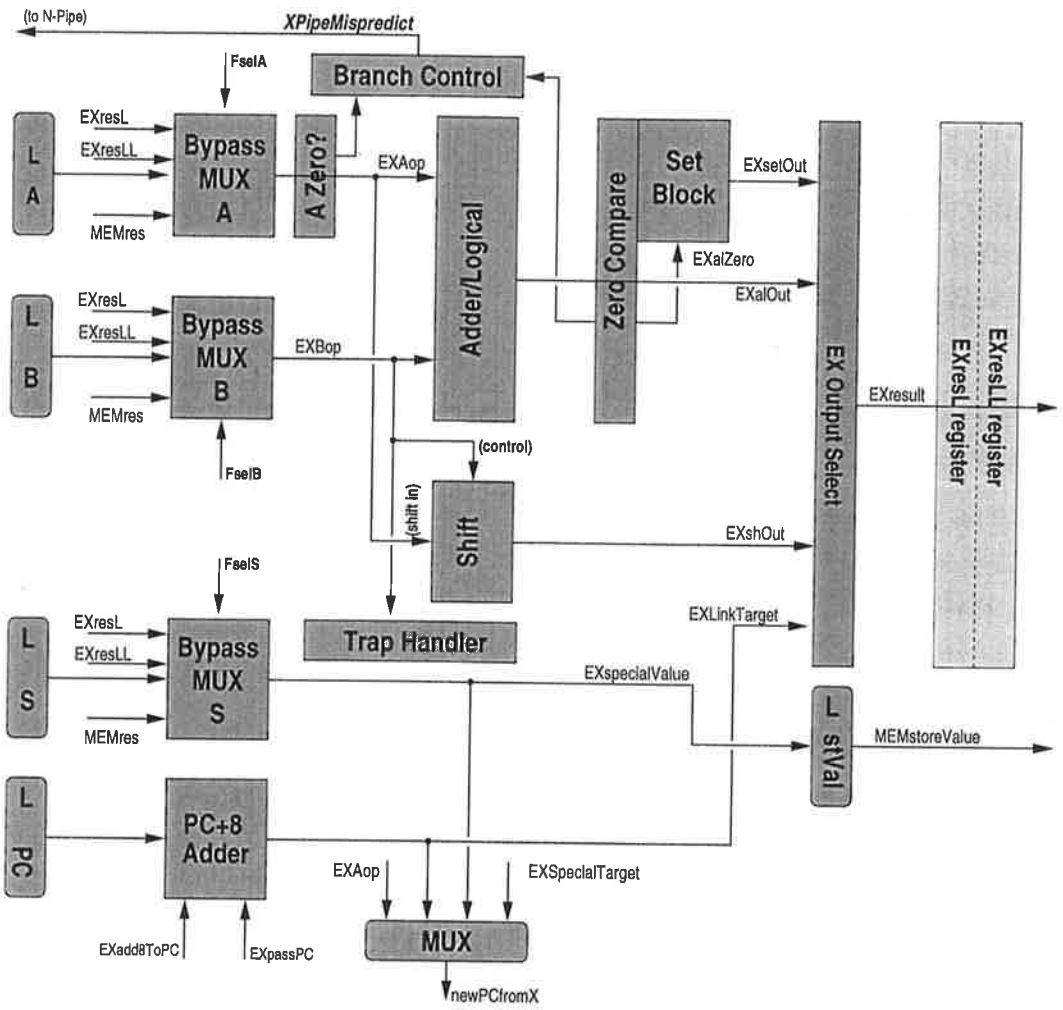


Figure 8.15 Amedo EX stage structure.

Integer ALU operations are straightforward, as operands are always on *EXAop* and *EXBop*, and the appropriate unit control signals (generated in ID) sequence the execution of the relevant instruction. The output of one of the three units (Adder/Logical, Set or Shift) is selected, and sent to *EXresult*. Memory access instructions are equivalent to an ADDU instruction in the EX stage (no overflow detection), calculating the address of the relevant memory location.

Conditional branches and unconditional linking instructions cause the *PC+8 Adder* to evaluate. Conditional branches cause the *A Zero?* unit to evaluate. This determines the outcome of the branch. If the outcome does not match the prediction, then the EX stage halts and drives either *EXspecialValue* (for a taken branch) or *PC+8* (for an untaken branch) onto *newPCfromX*, and asserts *XPipeMispredict*. There is no acknowledge from the N-Pipe, so this assertion is timed. The instruction produces no result, so it is *squashed*.

A JR instruction requires that the predicted target (on *EXBop*) be compared with the actual target (on *EXAop*). The ID stage sets up the *Logical* unit to do an *XOR*, and the logical output is compared to zero. If the result is not zero (and thus the predicted and actual targets are different), the EX stage asserts a halt and drives *EXAop*, the correct target, onto *newPCfromX* and asserts *XPipeMispredict*. The instruction is then squashed, and the halt removed. JALR instructions go through an identical sequence, but calculate the link address by evaluating the *PC+8 Adder* and sending this to *EXresult*, and are not squashed.

J instructions are perfectly predicted by the N-Pipe, and are simply squashed. JAL instructions cause the *PC+8 Adder* to evaluate, and its result to appear on *EXresult*, and are not squashed (although they are perfectly predicted).

It is possible that the time allocated to each branch instruction could be increased, eliminating the necessity of asserting a halt during misprediction. However, it is expected that branch prediction is effective, and misprediction should be the rarer case. Thus, the latencies of branch instructions are set for the time taken for each type of branch to determine that the *prediction is correct*, in addition to any final actions required for the branch (linking, for example). Misprediction will require that the stage halts and handles the condition by sending the correct result of the misprediction to the N-Pipe, involving a longer latency for the branch to complete.

A RTL description of the operation of the ID and EX stages of the X-Pipe is given in Appendix B.

Critical Path

The critical path varies heavily depending on the operation type. The basic component of the critical path for EX is the input latch and bypass multiplexer, and is then dependent on the type of instruction. The components of the critical path are shown in Table 8.2 on a per-instruction-type basis. This critical path data will be used to determine the speed of each instruction type in EX during the performance evaluation of the machine (used in Section 2.1.6 and Section 3).

Instn.	EXAop =0?	PC +8	Add (U)	Add (S)	Shift	Logical	EXout =0?	Set	O/P MUX
Add/Sub(S)	✗	✗	✗	✔	✗	✗	✗	✗	✔
Logical	✗	✗	✗	✗	✗	✔	✗	✗	✔
Shift	✗	✗	✗	✗	✔	✗	✗	✗	✔
Set(U)	✗	✗	✔	✗	✗	✗	✔	✔	✔
Set(S)	✗	✗	✗	✔	✗	✗	✔	✔	✔
Branch	✔	✗	✗	✗	✗	✗	✗	✗	✗
J	✗	✗	✗	✗	✗	✗	✗	✗	✗
JAL	✗	✔	✗	✗	✗	✗	✗	✗	✔
JR	✗	✗	✗	✗	✗	✔	✔	✗	✗
JALR	✗	✔	✗	✗	✗	✔	✔	✗	✔
MemOps	✗	✗	✔	✗	✗	✗	✗	✗	✔

Table 8.2 EX stage Critical Path Components.

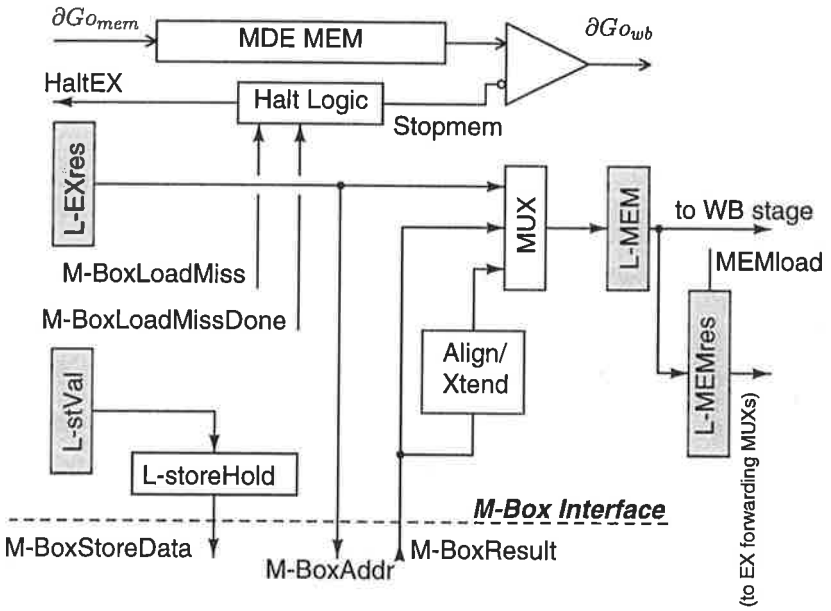


Figure 8.16 MEM stage structure.

2.1.3 Memory Stage

The structure of the MEM stage is shown in Figure 8.16, and is relatively simple, as most of the functionality embodied by the MEM stage is pushed into the M-Box (the Data Cache memory), which will be an implementation concern.

Instruction results arriving from EX which are not memory instructions simply pass through the output multiplexer and are sent to the writeback stage.

Loads send the address computed by the EX stage to the M-Box, which returns data on *M-BoxResult*. If the cache misses, it asserts *M-BoxLoadMiss*, halting the MEM stage and the rest of the X-Pipe. When the cache miss has been serviced and data to the MEM stage is valid, the cache asserts *M-BoxLoadMissDone* to restart the pipeline. In addition, because some loads are sub-word accesses which may require re-alignment to the correct position in the MEM stage result, as well as possible sign-extension (in the case of LB and LH instructions), subword accesses pass through the *Align/Xtend* block, causing a slightly longer latency for these loads than a normal word load (LW).

The MEM result register is only loaded during a load instruction. This ensures that the forwarding control of the decode stage knows that any load instruction recently issued (within the last two instructions) has its result in the *L-MEMres* latch for forwarding to the EX stage.

Store Operation

Arriving data for a store instruction is latched in the *L-storeHold* register. This allows the M-Box to determine the status of the store (hit or miss in the cache) *behind* the normal operation of the MEM stage. Figure 8.11 showed that the majority of store operations are not followed by load instructions, which would require preferential access to the MEM stage. In addition, the final write of the store into the cache memory can complete behind the cache check for a subsequent incoming store, meaning that store operation can be almost totally hidden, the exception being when a load instruction follows a store.

2.1.4 Forwarding and Stall Control

Forwarding is slightly more complex in Amedo than in the base machine because the pipeline control cannot rely on implicit synchronisation between stages. However, the forwarding paths in the microprocessor have considerably eased control because the stage which consumes the forwarded data is at most one stage away from the data source.

Figure 8.15 shows two result storage registers on the output of EX. The need for these two forwarding registers to bypass data can be illustrated by considering the following code fragment,

```
Forwarding Control:
   add     r1, ...     ; instruction 1
   sub     r2, ...     ; instruction 2
   sll     r1,r2       ; instruction 3, uses result of 1 and 2
```

In a synchronous machine, the result register of the MEM stage is used to forward the result of *instruction 1* to *instruction 3*. In Amedo, this is not possible. If *instruction 2* completes some time ahead of *instruction 3*, then the result of *instruction 3* will be in both the EX and MEM result registers of a synchronous machine, causing a forwarding error. Therefore, two result registers are used in EX, and the result register in MEM is only used for forwarding load data to the EX stage.

Forwarding control signals are generated using the structure of Figure 8.17. The incoming

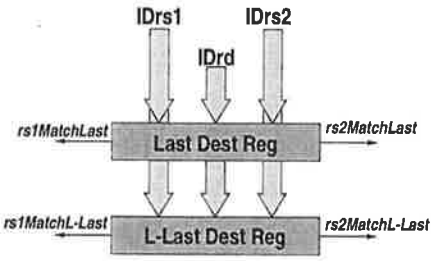


Figure 8.17 Forwarding Signal Generation.

`rs1` and `rs2` registers (if valid for this instruction) are compared to the last and last-last (or l-last) destination registers of previous instructions. If a match occurs, the type of instruction is then used to provide forwarding data for the multiplexers *Fmux-A*, *Fmux-B*, and *Fmux-S* of the EX stage in Figure 8.15. During a store instruction, the `rs2` input to the forwarding control logic is set to the field equivalent to `rd` in the store, and the forwarding outcome on `rs2` is used to setup forwarding on the *Fmux-S* multiplexer so that the store receives correct data.

When an instruction uses the result of the previous load instruction, a *load stall* results. This condition is detected by the logic of Figure 8.18. Load stalls halt the ID stage until the dependence on the MEM stage clears. The access of registers during a load stall must also be stalled. Consider the following code fragment,

```
Load Stall :
   lw      r6, ...
   lw      r7, ...
   add     r5,r6,r7  ; forwarding problem here as well as load stall
```

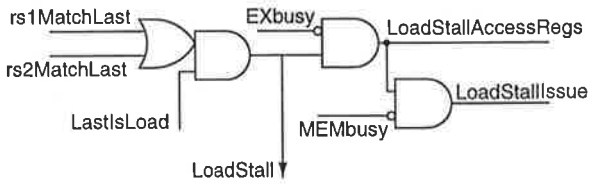


Figure 8.18 Load Stall Signal Generation.

In this case, a load stall occurs on the `add` instruction, and the `add` waits for the `lw r7` instruction to produce a result into the MEM load forward register. However, when the `add` arrives in ID and commences decode, the `lw r6` instruction is still completing in the memory stage, and if registers are sourced before the `add` stalls (due to the load stall), then an incorrect value of `r6` will be sourced. Thus, register access is stalled until the EX stage is empty, meaning that the `lw r6` instruction must have written its result into registers (an alternative is to use two forwading registers and paths from MEM, but this is an expensive solution to a rarely-occurring problem). When both the EX and MEM stages are free (and thus the stalling `lw r7` result is in the MEM load result register), the instruction issues with appropriate forwarding data added.

2.1.5 Free-Flow *Red* Timing Control

The X-Pipe uses a *Red* timing control mechanism to maximise throughput. The timing control structure for the X-Pipe is shown in Figure 8.19.

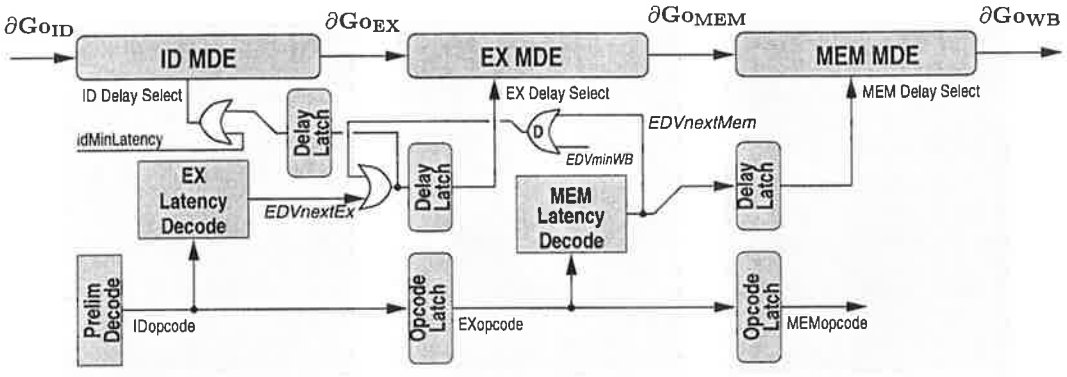


Figure 8.19 X-Pipe *Red* Latency Control. The latency of the WB stage is expected to be less than any other stage, thus a *Red* computation is not needed in MEM for the latency of WB. The added control elements required for pipeline *halt* are not shown here.

The generation of *Red* timing control is much easier than the general case, discussed in Chapter 6.3.1.3 (see page 149), because the X-Pipe is very short and only two stages do timing control calculations.

The EX decodes timing data for the next operation to occur in the MEM stage, and returns this value to ID through the *dynamic* OR used to hold the timing value, *EDVnextMem*. *EDVnextMem* must be either latched (in place of the *D*-OR gate) or held, and is evaluated when the decode of *EDVnextMem* is completed in the EX stage. The ISS stage for the X-Pipe, Instruction Decode, ORs this value with the calculated *EDVnextEx* value, providing this value as the latency of the EX stage on the next iteration. When the ISS issues, it latches the *EDVnextEx* value, and ORs it with the minimum latency of the ID stage to control the ID MDE element, used to control the issuing rate.

2.1.6 Preliminary Performance Evaluation

The use of the *Red* free-flow technique was expected to improve performance, but the level of this improvement was not known. The simulation model was thus enhanced with a model of the *Red* method, and the latencies of each operation in EX and MEM were set to what could be expected. EX stage latencies were based on Table 8.2. MEM latencies were the equivalent synchronous time of $\simeq$ 15ns minus a latch time. LW instructions do not require realignment and sign extension, and this was mirrored in the timing model. Pushing stores behind execution wherever possible, discussed in Section 2.1.3, was not implemented in this preliminary model.

The result of this change in timing characteristics is shown in Figure 8.20, and is based on the time required to execute the various benchmarks.

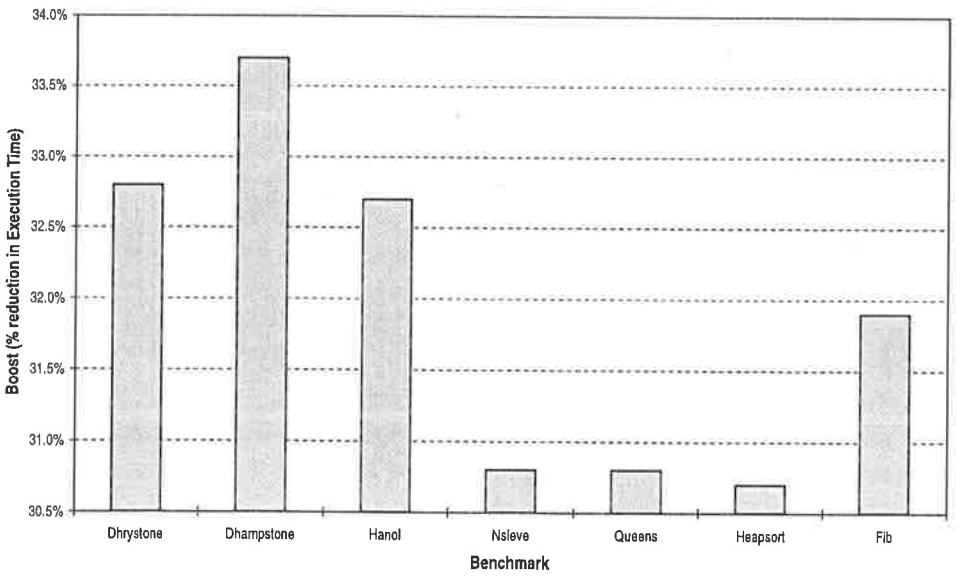


Figure 8.20 Amedo Performance using *Red* X-Pipe. This shows the reduction in time required to execute each benchmark on Amedo, compared to the base machine with a cycle time of 15.2ns. Note that the X-axis corresponds to a 30.5% reduction in cycle time.

This demonstrates that greater than 30% improvement in performance can be gained by using operation times that mirror the real execution time of the operation wherever possible. There are two factors constraining further performance gains. Both branch and load stall "cycle" loss cause roughly 10% performance drop each in the base machine. The performance penalty of these type of stalls in Amedo is higher because both branch mispredict and load stall delays are high compared to typical operation latencies. The second factor is the latency of the memory stage. Loads and stores account for a significant fraction of the dynamic instruction count (see Figure 8.4), and both of these operations have close to a full "cycle" latency (14s in 0.8μm CMOS). Some method of improving memory latency is required if continued performance gains are to be realised.

2.2 Load Register Buffers

Most techniques for improving the performance of the memory stage focus on using smaller, faster caches with prefetching [Dea92], or special structures to improve hit time in typical operations [Bra93] that operate in tandem with the traditional DCache architectures. These techniques do not improve load stall performance, because they still require an *address*, used as an index into the cache memory, to be delivered by the EX stage before beginning their access cycle. They do improve the average time to generate data in the MEM stage.

The approach taken here is to continue to use a small cache for data from the memory stage, but instead of indexing the cache based on the *address* of data, the cache is indexed by the data used to produce the address — the source register and offset. For example, in the instruction

```
lw r7,r1,#4      ;load r7 with word at address r1 + 4
```

the cache is indexed by the source register used (`r1`) and the offset (`#4`). When another load instruction arrives, the cache is checked to determine

- does the cache contain a *valid* entry for `r1`, and
- is the offset of this load compatible with the index of the cached entry.

If these both hold, then the cache *hits* and data can be sourced from this cache entry. This scheme is simple. The check can be performed in ID, and data can be received during the EX phase of this instruction (parallel to the address being computed), thus eliminating a load stall that might occur on the result of this load. In addition, benchmark traces show that relatively few registers are used as index registers for loads (see Figure 8.8), and thus a small amount of hardware could provide a significant boost in performance, as well as easing the memory timing bottleneck on the gain realisable by *Red* pipelining in Amedo. There are

disadvantages to this approach. Prefetching and updating will be required to get a reasonable hit rate. In addition, the number of usable cached registers is limited by the behaviour of the application. Furthermore, like all caches, it will sometimes *miss*, and the impact of the miss on performance should be minimised.

These small caches are named *Load Register Buffers* (LRBs), since they cache data on the index register used for the load. The data contained in a LRB entry is shown in Figure 8.21.

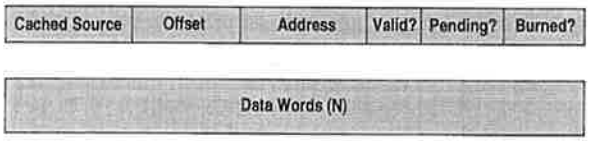


Figure 8.21 Load Register Buffer Entry Data Format.

The *Cached Source* field is a 5-bit field specifying the source register being cached, and the *Offset* field indicates the offset from this register that is in the cache. It is assumed that the LRB entry holds this offset, and *upwards* from this offset the next N words. The *address* of the LRB entry is not the full address, and is the address of the entry without the low order bits representing the number of words in the entry required, effectively specifying the address of the *first* word in the entry. This is required to maintain *coherence*, discussed in Section 2.2.1. The use of the *Valid?* field is straightforward. The *Pending?* field indicates that, although this entry is active, the data contained is not valid for some reason, and the entry is awaiting an update when the DCache becomes available. The meaning of the *Burned?* field will be discussed in Section 2.2.3.

Preliminary LRB Evaluation

The potential for LRBs as a DCache buffer and as a mechanism for reducing the incidence of load stalls was evaluated on the base machine. The LRB algorithm employed was not the final version used — any load instruction that writes to a cached LRB entry is invalidated, and *any* instruction which produces a result matching an LRB entry (excluding a load) is updated when using prefetch schemes, resulting in heavy prefetch memory traffic. Store coherence (see Section 2.2.1) was not considered in this evaluation.

The hit rate into various LRBs is shown in Figure 8.22. The timing model is still assuming a single-cycle memory access (the same as the base machine), so the effect of the interaction of the LRB with the memory hierachy was not factored into this experiment.

Several different prefetching algorithms were evaluated. The simplest is no prefetch, where

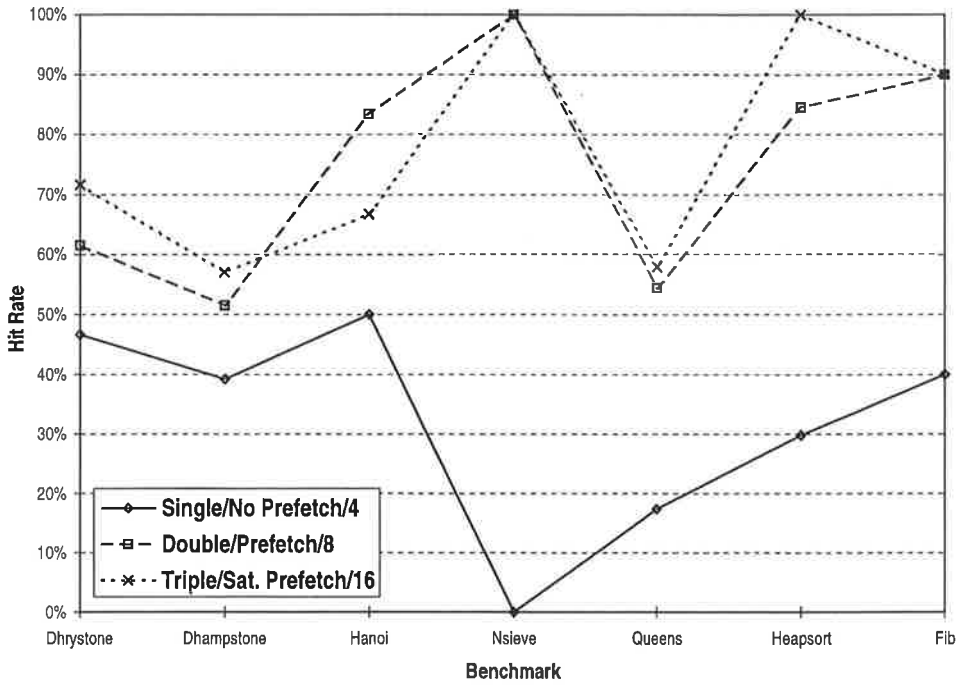


Figure 8.22 Load Register Buffer Hit Rates. The legend gives the number of entries in the LRB, the prefetch mechanism used, and the number of words in each LRB entry.

LRB entries are not updated when their cached source is updated, and are simply invalidated. This produced relatively poor results, no matter how wide (number of words stored) or deep (number of entries) the LRB. The second algorithm is a simple *prefetch* method. When the cached source is updated by a non-memory instruction, the LRB entry is updated. Any miss replaces the least-recently-used LRB entry. The final method considered is *Saturating-Replacement Predictive*, which is identical to the predictive method but uses a different method for replacement of entries when the LRB misses. Figure 8.8 shows that, although some benchmarks use relatively few source registers, some have a wider range of reference. The LRB should thus attempt to cache the *most often used* sources, and accept misses on those that are not used so often. This algorithm uses a saturating counter in each LRB to model access behaviour. When the entry causes a hit, the counter is incremented, limited by the saturation limit. If the entire LRB unit misses, all the entries counters' are decremented, and any that are at *zero* are candidates for replacement. This results in improved hit rate on some benchmarks which tend to use more source registers for loads (using a 2-bit counter for the saturating replacement counter).

The reduction in *load stalls*, where the instruction immediately following the load uses its result, causing a stall in the base machine, was then determined. The results are shown in Figure 8.23. Although the *Saturating-Replacement Predictive* method does not universally improve hit rate, it does improve load stall performance on *every* benchmark tested.

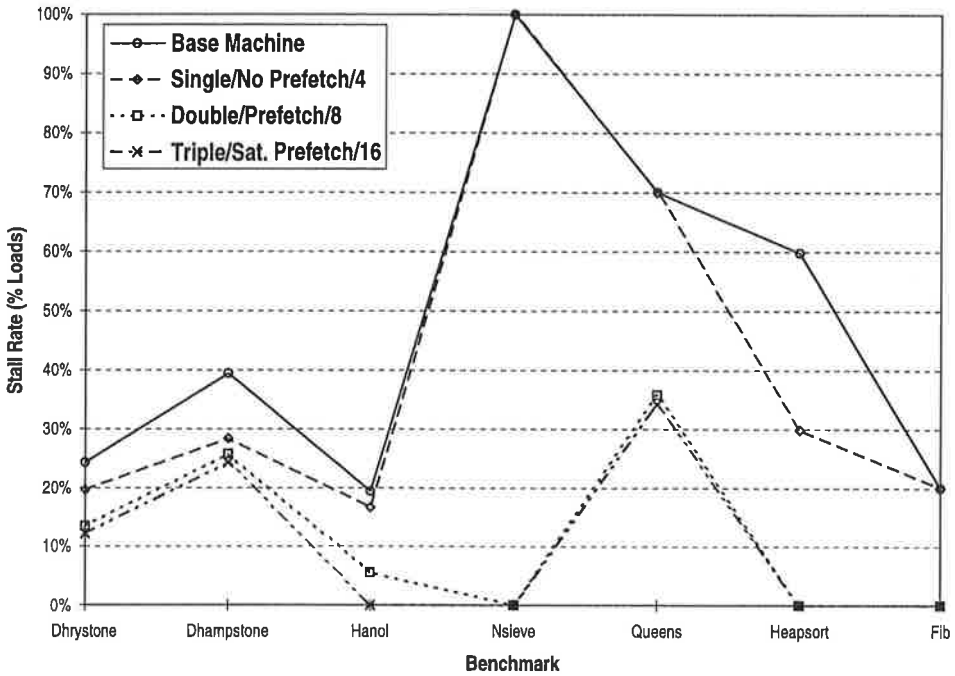


Figure 8.23 Load Stall Rates with Load Register Buffers. The legend gives the number of entries in the LRB, the prediction mechanism used, and the number of words in each entry of the LRB.

2.2.1 Coherence Issues

The LRBs are caching data produced by the DCache to improve hit rate and reduce load stalls, and must be kept *consistent* with the DCache and the state of the machine, and therefore *stores* to the MEM stage must be handled appropriately. As an example of consistency requirements, consider the following code fragment,

```
Coherence_Check :
   lhi     r1,(data_address >> 16)
   addui   r1,r1,(data_address & 0xffff)
   lw      r8,r1,# 0
   addui   r7,r1,# 8                    ; sets r7 to be 2 words above r1
   sw      r2,r7,#-4                    ; overwrites value in LRB(r1)
   lw      r9,r1,# 4                    ; coherence problem in LRB(r1)!
```

As the LRB is caching data on *source register*, there is a disambiguation issue in determining whether a store causes a coherence problem in a LRB entry, since two registers used as a load or store index register may contain the same or similar addresses (in this case, `r1` and `r7`). Therefore, a *store* in the MEM stage, in addition to performing a check in the DCache tags for a hit, must check for an *address* hit in the address each of the stored LRB entries, and update any that indicate a match. This will have no effect on the hit rate of the LRB,

and since this check and write access can occur while the store completes in the MEM stage, it causes no cycle time degradation of the store.

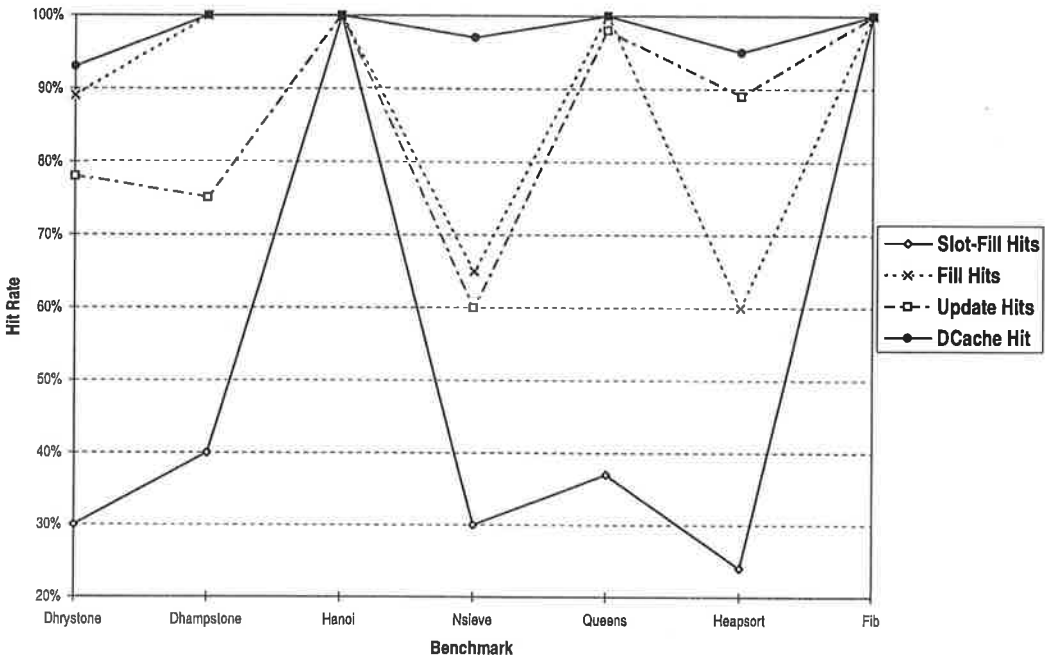
2.2.2 DCache Interaction

The initial evaluation of LRB units used the base machine assumption of perfect memory hierarchy, however, this is not the case in a practical machine. As LRBs do use heavy, self-initiated prefetching of data, a concern could arise that the LRB units are not going to provide good performance in practice because their unreasonable demands for prefetching may not be able to be met, and they could potentially miss often in the DCache, eliminating any performance benefit the idealised analysis showed.

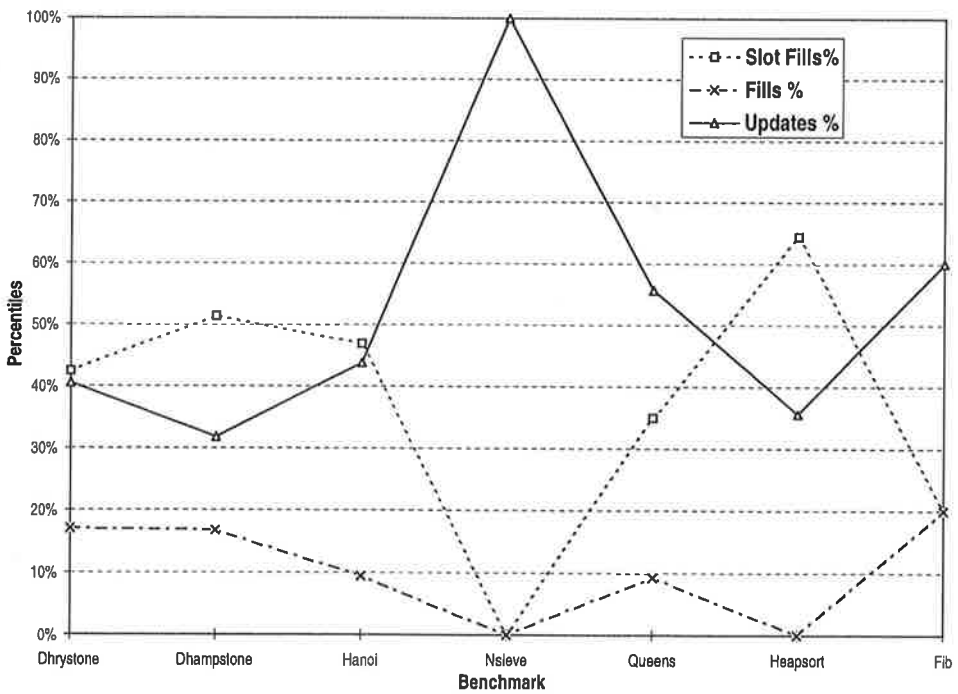
The accesses performed by the LRB were separated into three different classes. *Fill* accesses occur when the LRB misses and an entry is replaced. *Update* accesses occur when an *enabled* instruction updates the source register in a LRB entry. Only certain instructions cause these types of update accesses, namely those that are most likely to be adjusting the source register in preparation for a load. Other instructions simply mark the LRB entry as *pending*. *Slot* accesses are complex. The frequency of loads which overwrite an LRB entry (when the destination of the load matches the cached source register of an LRB entry) is relatively high. Therefore, when such a load occurs, the LRB entry is marked as pending. If a miss in the LRB occurs because this entry had not been updated then, *only* while this cached source register is in this LRB entry, this LRB entry is marked as *slot fill*. When an empty space appears in the memory stage (as the current instruction in MEM does not require a memory access), any LRB entries marked pending and *slot fill* enabled are attempted to be prefetched.

The hit rate of the LRB fill requests into the DCache, modelled as a 4Kbyte direct-mapped 32-byte block unit, was tested. The results of the experiment are shown in Figure 8.24.

This guides a number of choices in the design of the LRB prefetch and control algorithm. Figure 8.24(a) shows the hit rate in the DCache of the three kinds of prefetch accesses, and the hit rate of the DCache itself (on the same reference stream). This shows that some accesses are simply *useless*, since the DCache, with a hit rate close to 100% on all benchmarks, cannot provide the data requested by many of the prefetch requests, especially the *slot-fill* prefetch requests. If the LRB were designed such that it demanded the DCache fetch every access it requested, the memory traffic of the processor would explode upwards, and indeed most of this extra generated traffic would be useless. Figure 8.24(b) shows the relative frequency of the different kinds of access request from the LRB. *Slot fills* account for a large portion of the total generated traffic, but Figure 8.24(a) shows that many of these accesses are totally useless. The hit rate of *Update* and *Fill* accesses shows that they are more useful.



(a) Update Request Hit Rate in DCache



(b) Relative Frequency of LRB Fetches

Figure 8.24 LRB Interaction with DCache. (a) shows that, while the DCache hits at almost 100%, the prefetches initiated by the LRB do not hit in the DCache very often, and thus must not be generating useful data.

2.2.3 LRB Control and X-Pipe Interactions

The content control algorithm for the LRB entries is designed to maximise the utilisation of memory bandwidth, while attempting to minimise useless prefetch traffic from the LRB. In addition, the LRB now recalculates the stored offset when an entry is allocated during a load, so that the first LRB entry corresponds to the start of a DCache line. This is more realistic considering the DCache can only supply one cache line per cycle. The LRB control algorithm is described on a per-stage level.

The interaction of the LRB with the X-Pipe is complex, as to remove as many load stalls as possible, the LRB must be checked *early* for a hit, and appropriate information carried with the instruction down the pipeline to control the LRB interactions. Thus, the most complex LRB tasks are carried out in the ID stage. An overview of the integration of the LRB with the X-Pipe architecture is shown in Figure 8.25.

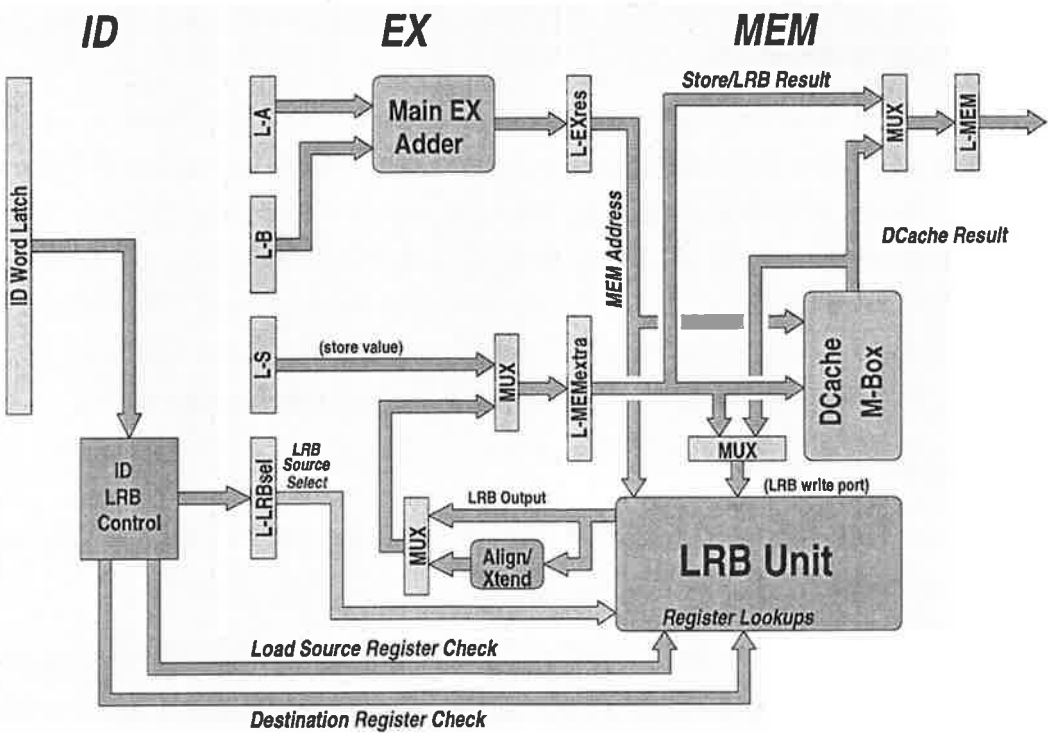


Figure 8.25 LRB Integration with the X-Pipe.

The ID stage performs all checks and updates concerning the non-data components of the LRB entries. The EX stage simply receives data based on the information produced by EX in the event of a load. The MEM stage produces two sources of data for the LRB entries. The first source is from stores, which may update the entries of one or more LRB entries when

they are in the MEM stage. The second source is LRB fills from the M-Box cache memory, requested either explicitly by the ID stage (for an LRB allocation on a load or an update fill on a relevant instruction) or due to a LRB-initiated prefetch during an empty MEM slot.

ID interface to LRB

The ID interaction with the LRB is described by Algorithm 8.1. The ID algorithm has two functions. The first is to determine if there is a hit in the LRB during a load instruction. If so, it generates access data to be used in the EX stage to receive LRB data. If not, the ID stage must determine whether to allocate a LRB entry to this load and which entry to allocate.

The second function is to check any instruction with a destination register that will change the machine state, and determine if the LRB is caching that register. If so, the ID stage determines whether this instruction should cause the LRB to be *updated* or simply *burned*, meaning that the contents are no longer valid.

LRB value use in EX stage

The ID stage has already determined if a LRB hit occurs on the present load — all that the EX stage has to do is receive data. The structure of this interface is shown in Figure 8.25. The EX stage does not *latch* the data from the LRB until the end of the EX cycle. This is so a store coherence update in the LRB, currently occurring the MEM stage, provides coherent data to the EX stage should a LRB match to the store have occured. The index into the LRB is provided by *hitIndex*, generated by the ID stage. Note that a duplicate *align and extend* block, for sub-word loads, will be required in the LRB forwarding path.

MEM interface to LRB

The MEM stage LRB control algorithm is described by Algorithm 8.2 for load instructions, and Algorithm 8.3 for non-load instructions.

A load instruction arriving at the MEM stage, with *loadAllocateEnabled* true, should read the DCache and write data into the LRB. So that the ID stage knows that the LRB is updated, the LRB entry that is being updated should be marked as $\overline{Pending}$ and $\overline{Burned}$ as soon as the load arrives in the MEM stage. In addition, if the destination register of the load matches a LRB entry, then the *result* of the load should be written to the LRB *address* field, and the entry marked as *Pending* and $\overline{Burned}$ — this enables an update to this LRB entry during an empty memory slot.

A store instruction in MEM demands an address lookup and compare for every LRB entry. Only the upper part of the address not pertaining to the index within the LRB is compared,

Algorithm 8.1 ID–LRB Interaction

```
if ID instruction is LOAD then
  Check load source reg. and offset for match in LRB entries
  if match then
    LRB Hit!
    update this LRB counter
    latch index to LRB for forwarding in EX into hitIndex
  else      — determine entry to replace
    Check 1 : any entry that is a valid source reg match ⇒ replace!
    Check 2 : any entry that is invalid
    Check 3 : any entry whose counter is zero
    if one check matches then
      label matched entry Pending and Burned
      latch the index to LRB in hitIndex
      loadAllocate is enabled on this instruction
    else      — a total LRB miss
      all LRB entries' counters are decremented
    end if
  end if
  if ID LOAD instruction has a matching rd in LRB then
    mark LRB entry as Pending and Burned
    latch index to LRB into updateIndex
    mark this instruction as loadDestMatch
  end if
end if

if ID instruction has rd that changes machine state then
  check rd for match in LRB  =  DestMatch
  check ID instruction in Enabled Update Instructions list  =  DestUpdateEnabled
  if DestMatch then
    if DestUpdateEnabled then
      mark instruction as loadDestUpdate
    end if
    mark entry as Pending and Burned
    latch matching LRB entry to updateIndex
  end if
end if
```

Algorithm 8.2 MEM–LRB Interaction for Load Instructions

if LRB hit in MEM **then**
  MUX out data on *MEMvalue* to *MEMresult*
else if *loadAllocate* is enabled **then**
  immediately mark LRB entry indexed by *hitIndex* as $\overline{Pending}$ and $\overline{Burned}$
  access DCache to obtain data for LRB
  compute LRB offset that aligns cached source with DCache line size and load into LRB
  MUX out load data as normal
else — not a LRB hit, and an entry is not allocated for this load
  operate DCache as normal
end if
if *loadDestUpdate* true **then**
  latch MEM result to stored address of *updateIndex* indexed entry
  mark indexed entry as *Pending* and $\overline{Burned}$.
end if

and if a match occurs in any part of any LRB, the result of the store is written to the entry. This takes place concurrently with the normal store access in the MEM stage.

If the instruction is *updateEnabled* (when the destination register matches a LRB entry, and this instruction is typically used to update a memory index register), then the DCache is accessed at the address indicated by the instruction result. If a DCache miss occurs, the miss action of refilling a line in the DCache memory is suppressed, and the update ignored. However, if a hit occurs, the LRB matched entry is updated and marked $\overline{Pending}$ and $\overline{Burned}$, and is then valid. If the instruction does nothing in the MEM stage, then an update to any *Pending* but not *Burned* LRB entries is attempted.

Update Suppression

Since there is a two-stage delay between the generation of LRB update signals and the actual updating of the entry, the LRB control must ensure that the LRB is not updated falsely. LRB updates are suppressed in the MEM stage when ID or EX has a matching *updateIndex* field, so that false updates do not occur. An example of false updating is

```
False Update :
  add     r1, ...     ; executing in MEM, updates LRB entry for r1
  mult    r1, ...     ; burned LRB field in ID, but in EX needs to suppress
                      ;   updating of r1 in LRB during MEM access
```

Algorithm 8.3 MEM–LRB Interaction for Store and Non-Memory Instructions

if MEM instruction is STORE **then**
  check all LRB entries for matching address, with $Valid \cdot \overline{Pending} \cdot \overline{Burned}$
  if match(s) detected **then**
    update matched LRB entries with store data
    DCache STORE access proceeds as normal
  end if
else if *updateEnabled* is true **then** — MEM has update instruction
  access DCache and suppress handling of any resultant miss ⇒ *DChitU*
  if *DChitU* **then**
    matched entry indicated by *updateIndex*
    mark entry as $Valid \cdot \overline{Pending} \cdot \overline{Burned}$
    latch result of MEM instruction as LRB entries' *stored address*
    calculate offset to align stored offset to give DCache line alignment
  else
    do nothing (entry is already not usable)
  end if
else — *Empty* slot in MEM stage
  if any entries are $Pending \cdot \overline{Burned} \cdot Valid \Rightarrow slotEnable$ **then**
    attempt DCache fill from first matched entries' stored address
    if DCache Hit **then**
      mark entry as $\overline{Pending}$
      calculate offset to align stored offset to give DCache line alignment
      load LRB entry with DCache data
    else — DCache miss on update
      mark entry as *Burned*
    end if
  else — No LRB entries require an attempted slot fill
    do nothing
  end if
end if

In this case, the `mult` instruction in EX suppresses any updating of the `r1` LRB entry due to the `add` instruction, which would give an incorrect result if allowed.

X-Pipe Timing Control

The use of a LRB requires that the timing control mechanism for *Red* pipelining be modified. If a load stall is detected along with a previous LRB *hit*, then the load stall is eliminated because data can be forwarded through the EX stage. A LRB hit also takes minimal time in the MEM stage, because nothing is done (unless the destination of the load hits a LRB entry, updating the entries' address). However, the *prefetching* accesses in the MEM stage slow the MEM stage if a subsequent instruction requires memory access (when the instruction in EX is a store or a load that missed in the LRB).

2.3 N-Pipe

The function of the N-Pipe is to fetch operands for the X-Pipe, and to provide a predicted flow of addresses that attempts to mirrors the path that will be taken by the X-Pipe, thereby minimising time loss from mispredicted control transfers. In addition, because the X-Pipe is likely to consume operands at a rate *faster* than the base machine cycle time (if FeFA *Red* pipelining is used), bandwidth issues must be considered in the structure of the N-Pipe.

The structure of the N-Pipe is shown in Figure 8.26.

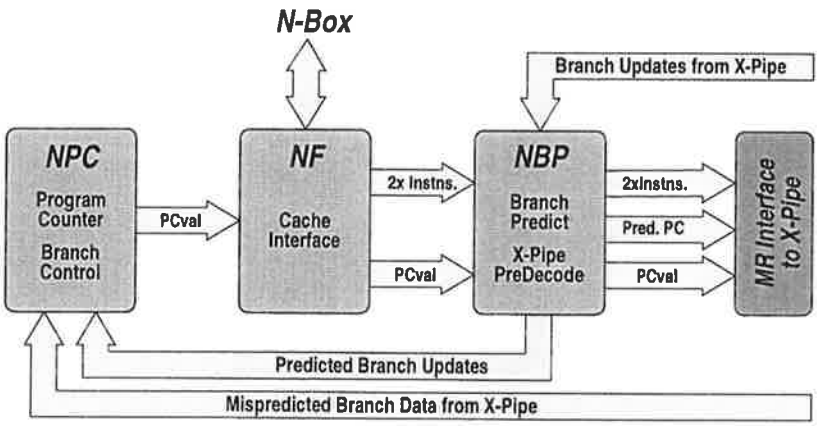


Figure 8.26 N-Pipe Structure.

The N-Pipe Program Counter stage (NPC), controls the fetch address, reloading the program counter from either the N-Pipe Branch Predictor (NBP) stage, or preferentially the address provided as a *mispredicted* branch target from the X-Pipe.

The Fetch Stage and Fetch Width

The Fetch stage (NF), interfaces to the cache box, the *N-Box* of Figure 8.12, to fetch *two* instructions per requested fetch. A fetch width of two *doubles* the fetch bandwidth, while allowing a single-stage simple branch prediction mechanism. Using a wider path would demand that the branch predict stage contend with potentially more than one branch in a fetch block, complicating the control and the datapath.

Fetches are *always* aligned on a two-word boundary. This results in one useless, squashed instruction being fetched when the program counter changes to an address not aligned on a two-word boundary. Using a fetch aligned in this way considerably eases the N-Box design, since fetches will always be aligned on a cache line, and also eases the generation of PC addresses for the two produced instructions before they are inserted into the buffer between the N-Pipe and X-Pipe. Fetching two instructions also improves utilisation of fetch bandwidth (done in the ECSTAC design in a different manner).

The first instruction in the two-word fetch is *Slot F* (First Slot, with address `--000`), with the second being *Slot S* (Second Slot, with address `--100`).

2.3.1 Branch Prediction

The branch predictor for the N-Pipe must handle three different classes of branches,

- Unconditional Static Target (UST) Branches, J and JAL,
- Unconditional Dynamic Target (UDT) Branches, JR and JALR, and
- Conditional Static Target (CST) Branches, BNEZ and BEQZ.

UST branches are *perfectly* predictable, as the target address and taken status of the branch can be determined without any need to run the instruction through the execute stage of the machine.

UDT branches are more difficult to handle, since an update of the register used to generate the target may be pending in the pipeline, and consistent register operands are only available in the X-Pipe. UDT branches generally fall into one of three categories,

- Procedure returns,
- Procedure calls whose address may vary dynamically, and
- Procedure calls from a list of procedures (for example, the *switch/case* statement in C).

Procedure returns are relatively easy to predict with DLX as the return register is fixed by the ISA, and it is available for general manipulation. Using a *stack* for subroutine returns is very effective in computing return addresses for procedure returns [PH96]. Procedure calls

from lists are virtually impossible to predict, since the target address is contained in memory and will not be loaded until very shortly before the branch, and the dynamic target is likely to change often. An attempt to predict both of these types can be made by providing a dedicated access port to the register file (since this is the final location of the target address). This works well when the procedure is passed as an argument, as it is typically available a number of instructions before the branch executes.

The target of CST branches can be determined exactly in the prediction stage, however the conditional status of the branch is not available. One common method for predicting the direction of the branch is a *branch history table*, a memory indexed by the lower address bits of the instruction that predicts the outcome of the branch [PH96], typically based on a saturating counter. This simple approach to prediction is adopted here. The NBP always calculates the target of the branch, predicted taken or not.

The structure necessary in the N-Pipe Branch Predictor (NBP) stage to implement these functions is shown in Figure 8.27. Only one of the two instructions in the slot can be a

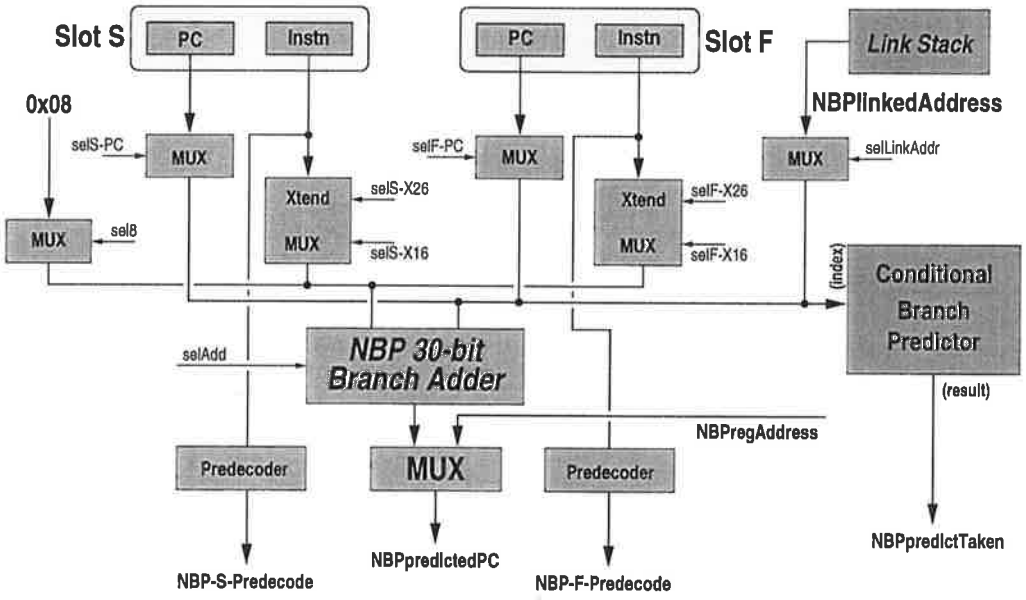


Figure 8.27 N-Pipe Branch Predictor (NBP) Datapath.

branch (since all control transfers are defined to have a branch delay slot instruction [SK96a]). Calculating the target of UST branches requires adding the long immediate field in the J-type instruction format to the current PC address, and then adding 4 (since index-based control transfers are defined to be relative to the branch delay slot). This may seem like it requires

two adders, or one adder and a compressor. However, instruction addresses are only 30 bits long (the bottom two are irrelevant as instructions are word quantities), and the adder can change the carry in to the 0^{th} bit for these calculations to effectively add 4 to the instruction address so calculated.

During JR and JALR instructions, the target comes from one of two sources. If the register jump address is `r31` (a probable procedure return), the address is taken off the Link Stack. If the instruction does not use `r31`, the source is accessed from the register port provided to the NBP and returns via *NBPregAddress*, to be used as a predicted target.

Calculating the target of CST branches is similar to the UST case, however, only the bottom 11 bits of the instruction word are used as an offset, and the status of the branch must be predicted via the *Conditional Branch Predictor*. Any taken control transfer stalls the N-Pipe and updates the program counter in NPC.

The link stack holds program positions linked to by the instructions JAL and JALR. When a linking instruction arrives at NBP, the current PC is loaded into the link stack, implemented as a rotating ring FIFO like that used for buffering in free-flow. When an instruction arrives at NBP which uses the link register (`jr r31` or `jalr r31`), an entry is removed from the link stack and sent to the adder. As the process of *linking* only stores the address of the instruction which caused the link, and the program must return to the instruction after the branch delay slot of the linking instruction, the value 8 must be added to the removed address to set the link address properly. The link stack interface is shown in Figure 8.28.

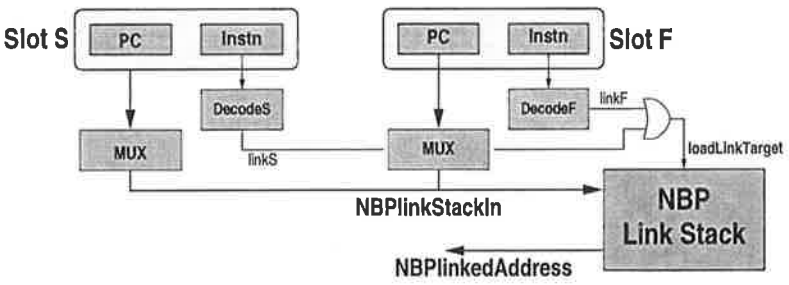


Figure 8.28 N-Pipe Link Stack Interface.

The depth of procedure calls was not evaluated in the experiments on the base machine, but experience suggests that a 4 to 16 entry buffer provides almost perfect prediction for procedure returns [PH96].

Asynchronous Execution and Branch Delay Slots

In this model of execution, a taken branch instruction in NBP *Slot S* coupled with the NF stage halting due to a miss can cause a potential problem in the machine control to surface. The branch instruction will execute in the X-Pipe long before the cache miss completes, and the X-Pipe squashes all instructions in the N-Pipe if a misprediction occurs. However, branch instructions are defined in DLX as executing a delay slot instruction. This can lead to erroneous behaviour if the branch delay slot is not in the X-Pipe (namely instruction decode) when the branch executes.

Two methods exist for resolving this potential hazard. One is to demand that branch instructions never appear in the last word of an instruction cache block. This is implementation-specific, and will not scale if the instruction cache block size is decreased, and might also cause difficulties for dynamic linkers. The other, used in Amedo, is to stall the assertion of the *HaltX* signal, and the final processing of any branch misprediction in the execute stage of the X-Pipe, until an instruction arrives at the decode stage, which must by definition be the delay slot instruction.

Branch Prediction Accuracy

Using the existing simulator, branch prediction was added to the model to determine the effect of branch prediction. The success of the predictors used is shown in Figure 8.29.

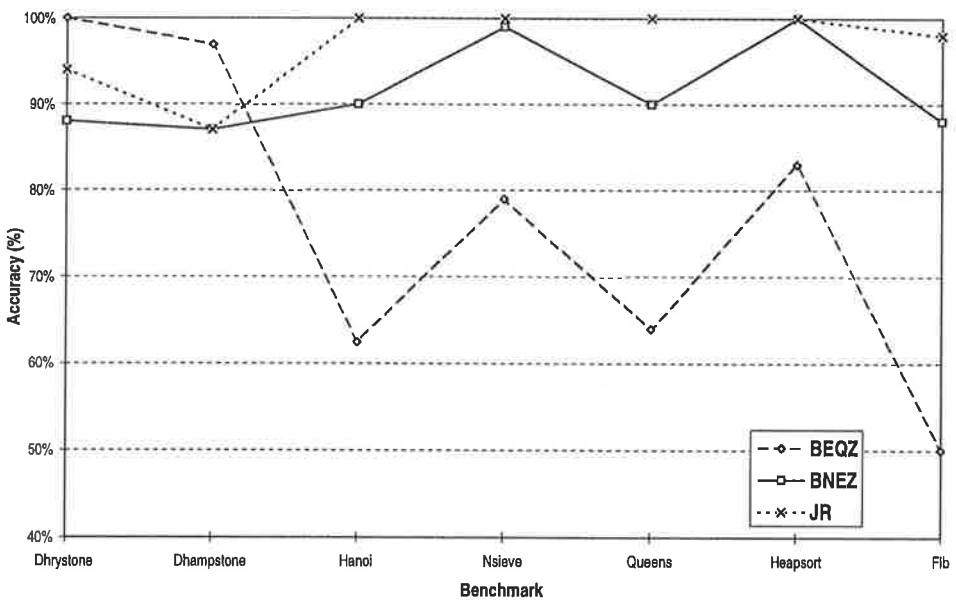


Figure 8.29 Branch Predictor Performance. A 256-entry 2-bit saturating counter was used for conditional branch prediction, with a 10-entry subroutine return address stack and a register read port providing prediction for JR and JALR instructions. The size of the conditional branch predictor has little effect on prediction accuracy for these benchmarks.

The prediction success for BNEZ and JR instructions is excellent. However, the performance of BEQZ conditional branches is disappointing at around 70%, and simply increasing the size of the predictor has no effect on the prediction accuracy. None of the preliminary benchmarks evaluated used the JALR instruction.

2.3.2 Halt Control

Halt control in the N-Pipe is complex compared to the simple schemes considered in Chapter 6. The interface buffer can halt the N-Pipe if it becomes full. The NBP halts the N-Pipe if it needs to update the NPC to load a predicted taken branch target. The NF stage can halt if it encounters an *address error* exception (the control for this will not be considered here), and halts the NPC if a cache miss occurs. Finally, the X-Pipe can halt the NPC if a branch target is mispredicted, leading to the entire N-Pipe being squashed. These sources of halts, and the control signals used, are shown in Figure 8.30.

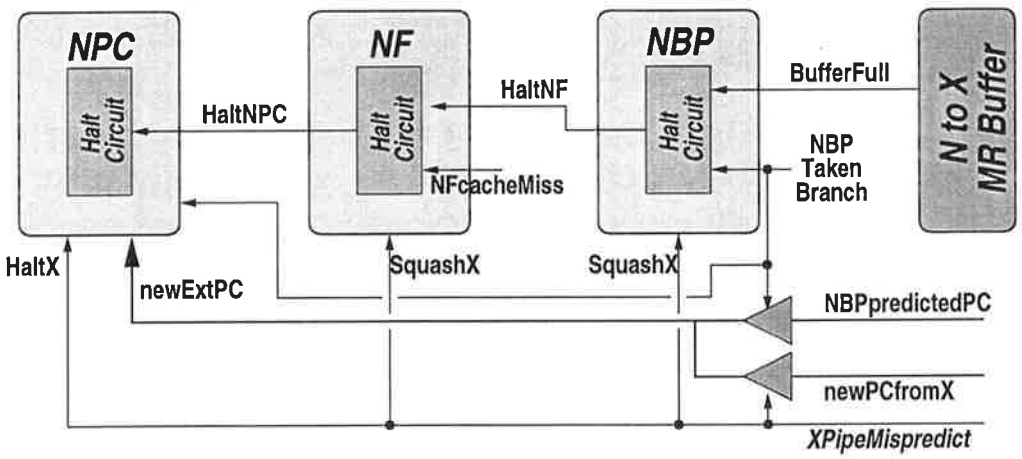


Figure 8.30 N-Pipe Halting Sources.

2.3.3 Program Counter Stage

The program counter stage, NPC, has two sources of updates, an update due to taken branch from NBP, and an update due to a mispredicted branch from the X-Pipe.

A NPC halt can be asserted from two sources. The NBP stage halts the entire pipeline and waits until the signal *HaltNPC* is asserted before reloading the NPC. However, the mispredict update from the X-Pipe does not halt the N-Pipe (it squashes the contents of the N-Pipe), and is uncorrelated to the present status of the NPC stage. The assertion of *HaltX* kills all operands in the N-Pipe, and thus any update occurring through *HaltNPipeBT* due to a taken branch is killed when *HaltX* is asserted.

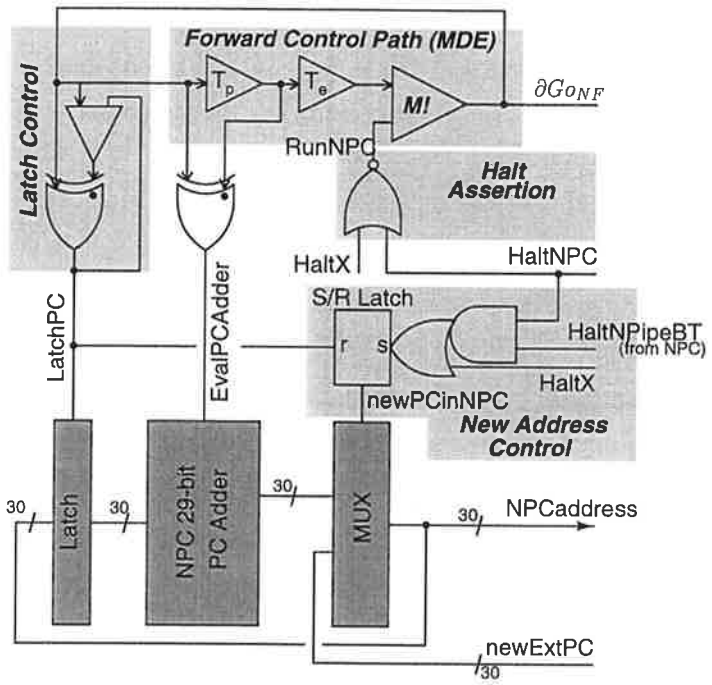


Figure 8.31 N-Pipe Program Counter (NPC).

Either halt occurring sets the multiplexer to receive data from *NewExtPC*, loading this value onto *NPCaddress*, which goes to the NF stage. When the halt is deasserted (either through $\downarrow HaltX$ or $\downarrow HaltNPC$), the output event ∂Go_{nf} will be issued. The output *send* gate marked *M!* represents a potential metastability failure point because of the arrival of the *HaltX* signal is a broadcast-type halt (see Chapter 6.2.3.1). However, no special measures are adopted here for metastability hardening.

Once issued, the NPC loads its input latch with the new PC value and increments on an eight-byte boundary for the next fetch (which must be aligned to an eight-byte boundary). This also de-asserts the signal *newPCinNPC* controlling the multiplexer.

Response Time Issues

The X-Pipe is decoupled from the N-Pipe, and when a misprediction occurs in the X-Pipe, it simply drives the address onto *newPCfromX*, asserts *XPipeMispredict* for some finite time without handshaking, and then starts processing again. This can cause a problem with the datapath of Figure 8.31 if the cache is currently in the *miss* state. The NPC is certainly reloaded, but it may be stalled waiting to issue to the NF stage. If this process involves a long-latency fetch from external memory, then the address on *newExtPC* may be corrupted (since it is not being driven anymore, as *XPipeMispredict* only pulses). A solution is to implement a weak charge-holding circuit on *newExtPC* when *newPCinNPC* is high, holding the value of the bus *newExtPC*.

2.3.4 Pseudo-Red Timing Control

The entire prediction process can be bypassed if neither of the *Slot* instructions are branches. Indeed, this will be the case relatively often due to the size of the basic block in benchmark code (see Figure 8.6).

If the NBP stage causes a cycle time degradation in the N-Pipe during prediction, a pseudo-*Red* variable latency technique can be employed to mitigate the effect of the delay of the NBP during non-branch processing. The Matched Delay Elements (MDEs) in the NPC and NF stages can be made *two-valued*. The smaller value represents the normal delay of the NPC/NF stages (most likely to be the access time of the NF stage), with the additional delay being the added delay necessary to process a branch in NBP.

When NBP receives the two-word packet from IF, if a branch is present it can assert a signal *global* to the N-Pipe that switches the MDE values to the higher value. This is safe as the other stages can only have *just* started processing. However, in the more common case where branch processing is not required, this signal would not be asserted allowing the stages to run at the fetch rate, not the predict rate.

In the current architecture, the branch processing step is relatively simple and will be faster than the fetch stage. Therefore, this pseudo-*red* mode of control is not required, however it is used in the NBP stage to issue the two instructions to the buffer faster if neither are branches. However, if the NBP stage became more complex (due to a wider fetch width or more complex branch processing), this technique will prove useful.

3 Amedo Performance

The performance of the full Amedo processor was evaluated using the benchmarks of Table 8.1 in addition to some additional programs described in Table 8.3. The final model of the Amedo processor employed the following features

- Full FeFA *Red* pipelining,
- Load Register Buffers (4-entry, 8 words per entry, saturating counter),
- Full DCache timing model that factors miss time,
- Branch prediction using a 256-bit conditional predictor (see Figure 8.29), a 10-entry return address stack, and an additional register file port for predicting JR/JALR jumps,
- 4-KByte direct-mapped DCache with 32-byte blocks.

The effect of the ICache was not factored in these results. The ICache was modelled an earlier experiment, and showed that for any ICache of size more than 2-KBytes, most of the program code fits in the ICache completely, and thus the only effect modelled is the transient fill-up of the ICache. However, the memory system outside of the DCache was not modelled (and is still idealised).

Name	Description
Quick	Quicksort algorithm on large array of randomised integers
Fact	Factorial computation
String	Boyer-Moore-Horspool string matching
DesCrypt	DES code encryption

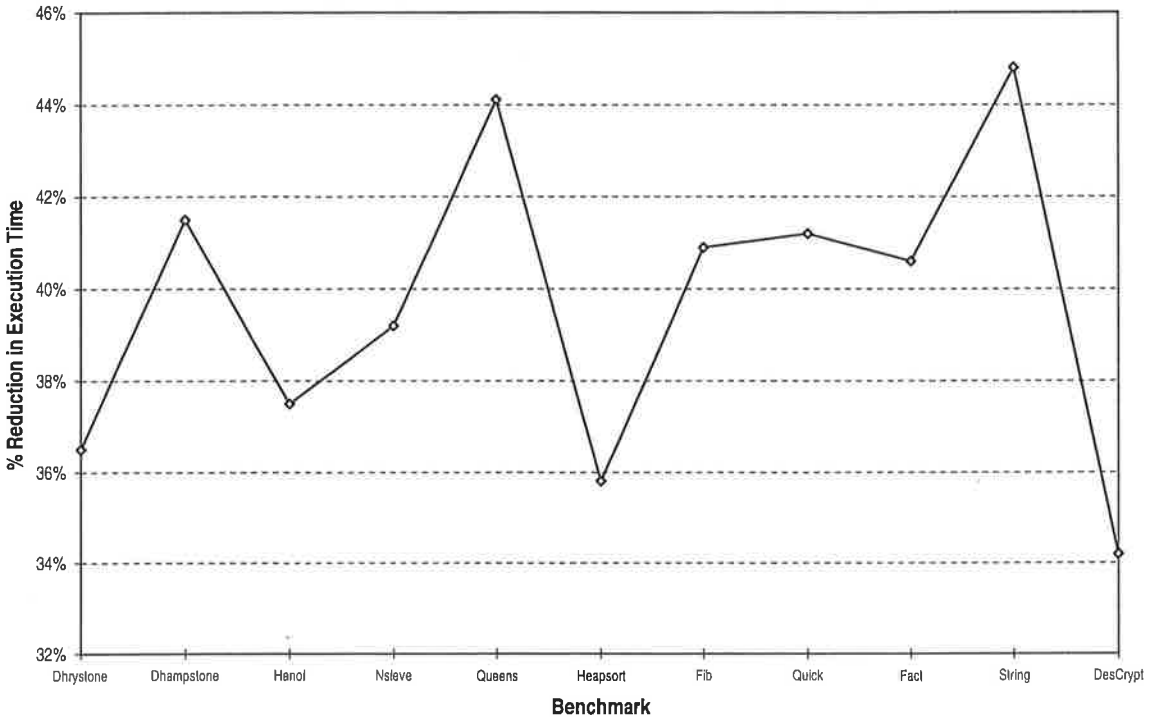
Table 8.3 Additional Benchmark Programs. Additional relatively small integer programs were used to provide more detailed performance figures for Amedo.

The timing of operations in the simulation of Amedo is shown in Table 8.4. The timing of EX operations is based on Table 8.2, using parameters from the 0.8μm CMOS process used in ECSTAC [PP94]. The Instruction Decode time for a miss in the LRB is longer than the normal ID latency because the ID stage must determine what entry to replace in the LRB, if any.

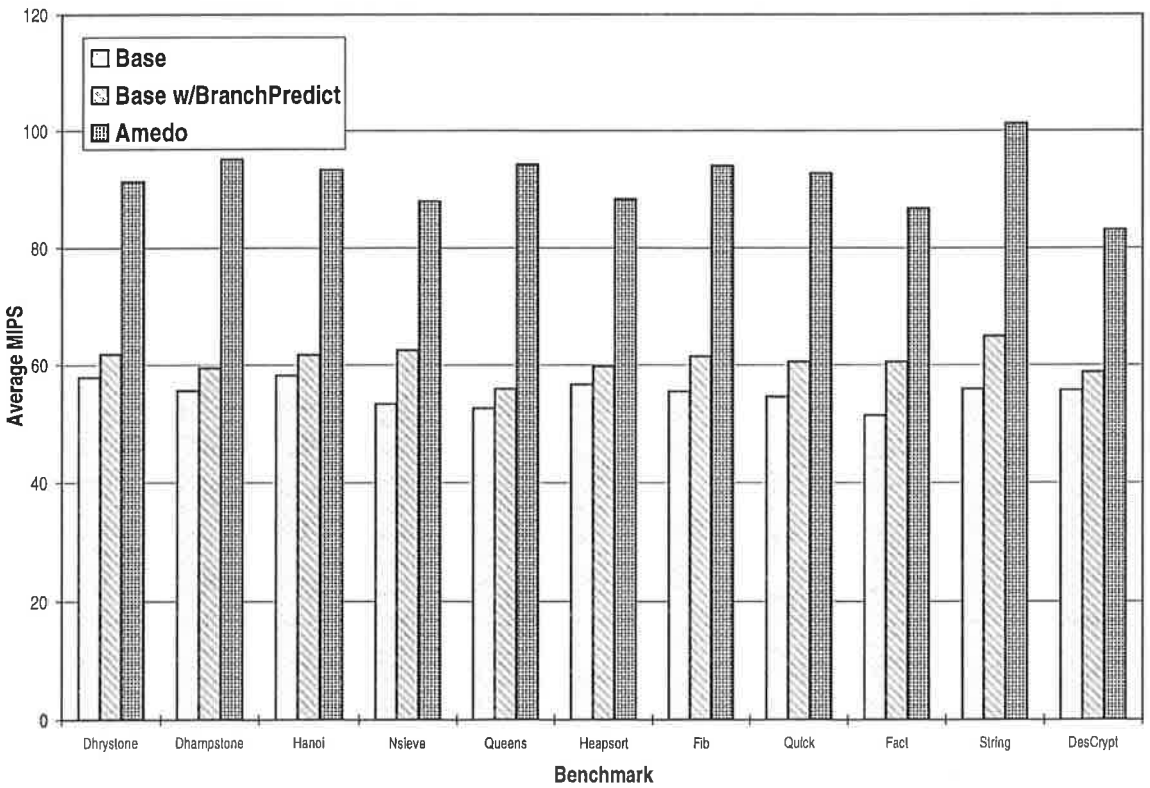
Branch mispredictions take an extra 4ns in EX to drive the address to the NPC stage and pulse *XPipeMispredict*. When the mispredicted target arrives at ID, a 15ns penalty for the branch is added. Furthermore, if either of the instructions in the first *slot* fetched by the N-Pipe are branches, the penalty is taken to be 23ns due to the required time for branch prediction in the N-Pipe.

The simulator had the full LRB functionality and control algorithm implemented (see page 227). The final performance of these benchmarks on Amedo is shown in Figure 8.32 compared to the base architecture performance.

To determine the relative performance of the heavily optimised memory system, the base machine average MIPS measure (with branch prediction) was scaled by 30% to account for the approximate effect of the *Red* technique (based on Figure 8.20), and the performance gain of Amedo over this scaled figure was determined. This comparison is shown in Figure 8.33. This shows that the gains in performance do not come wholly from the use of *Red* pipelining — from 20% to 40% greater performance is being delivered by other sources, namely the



(a) Reduction in Execution Time of Amedo with respect to the base machine



(b) Average Equivalent Integer MIPS

Figure 8.32 Amedo Performance.

Stage	Instn.	Time	Comment
ID	Load	10ns	if not LRB hit
	Load	6ns	if LRB hit
	others	6ns	
EX	Add Unsigned	8ns	
	Add Signed	12ns	
	Set	12ns	
	Branches/Jumps	6ns	if prediction correct
	Linked Jumps	8ns	if prediction correct
	Mispredict	+4ns	time to reload PC
	Logical/Shift	6ns	
	Load Stall	op+MEM time	add MEM time to EX time
MEM	Load Word	12ns	
	Load Non-Word	14ns	
	Load Miss	50ns	external access time
	Store	8ns	
	Store	12ns	if next instn is store or not-LRB-hit load
	LRB hit	6ns	
	LRB prefetch	12ns	only if next instn is store or non-LRB-hit load
	(others)	6ns	
ID	Mispredict	15ns	mispredict penalty when target in ID
		23ns	if either of first slot is branch (need prediction)

Table 8.4 Amedo Timing Parameters.

memory system and its timing methodology. Note that the improvement is generally greater on those benchmarks which were not heavily affected by issues surrounding the LRB (see Section 3.1).

Characteristics of the branch prediction mechanism are shown in Figure 8.34. The performance gain realised from branch prediction is obviously critical in improving performance on both the base and Amedo machines. The characteristics of the LRB are shown in Figure 8.35, and show some very distinct features.

3.1 Load Register Buffer Optimisations

The performance of the LRBs on certain benchmarks, shown in Figure 8.35, is clearly problematic. These programs tend to have small inner loops which perform relatively few memory instructions per loop, but execute the loop a number of times. The reason for this perfor-

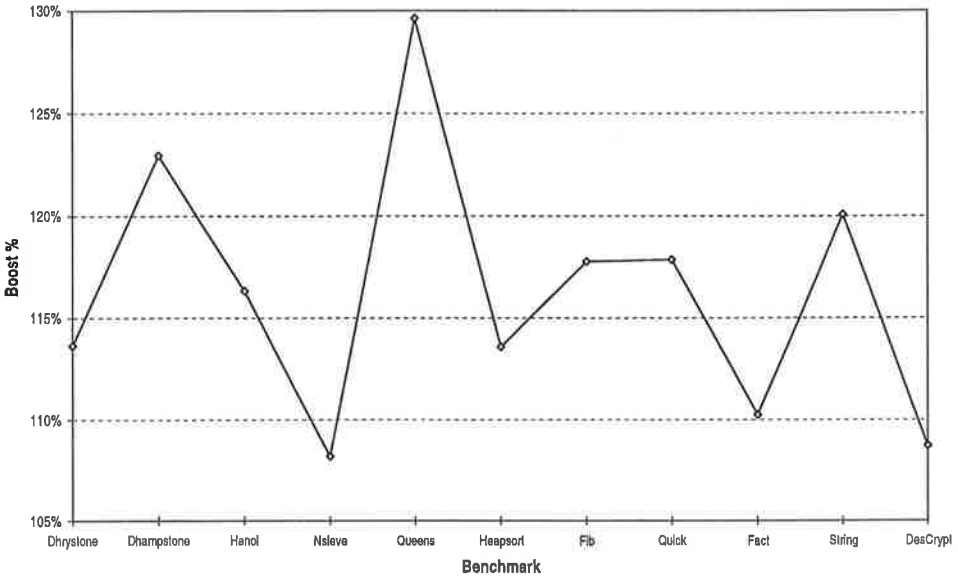


Figure 8.33 Scaled Amedo Performance Comparison.

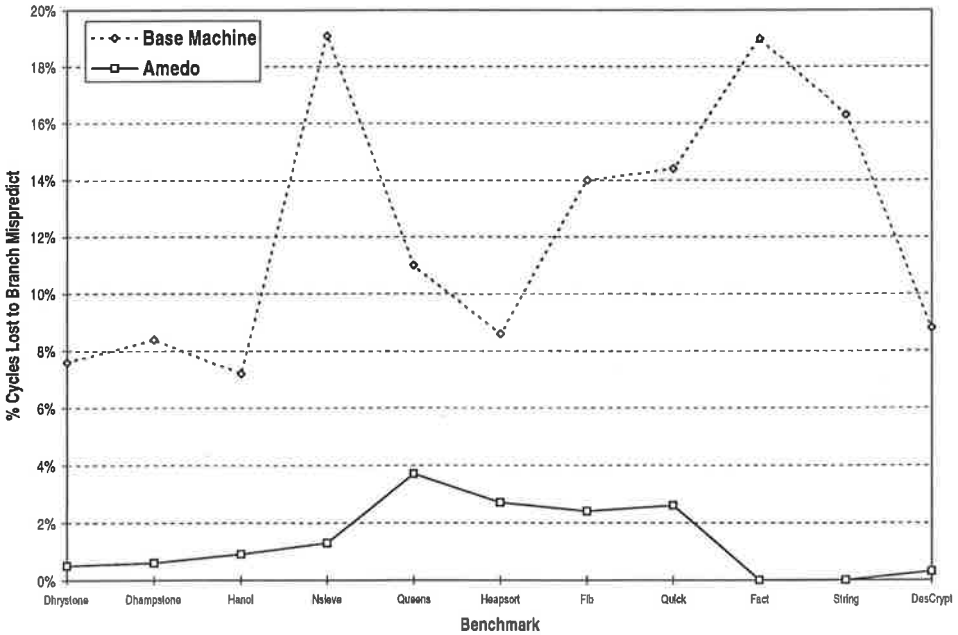
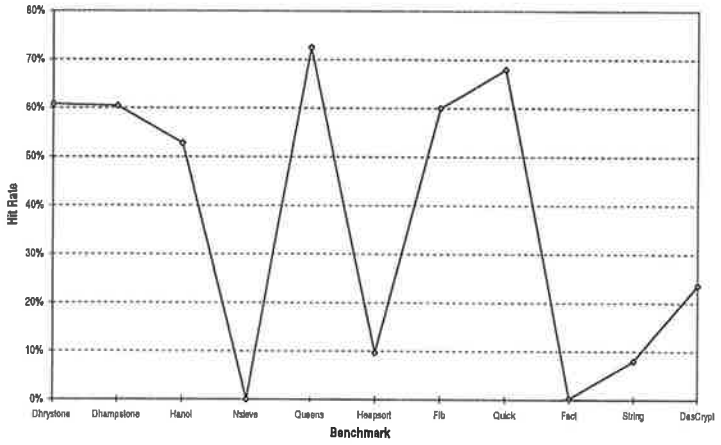
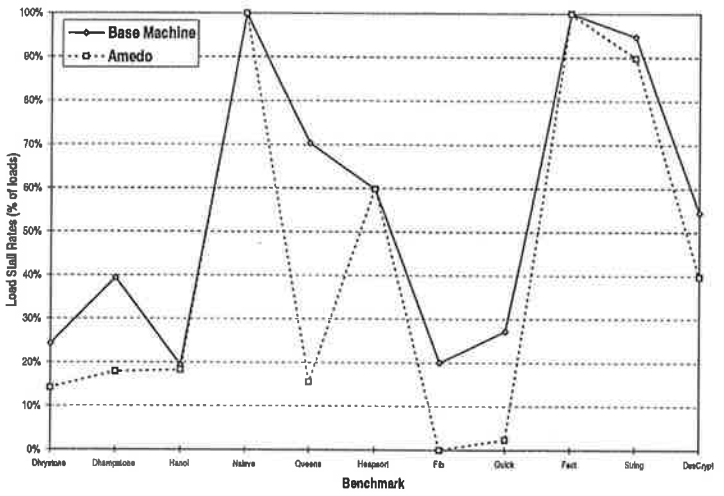


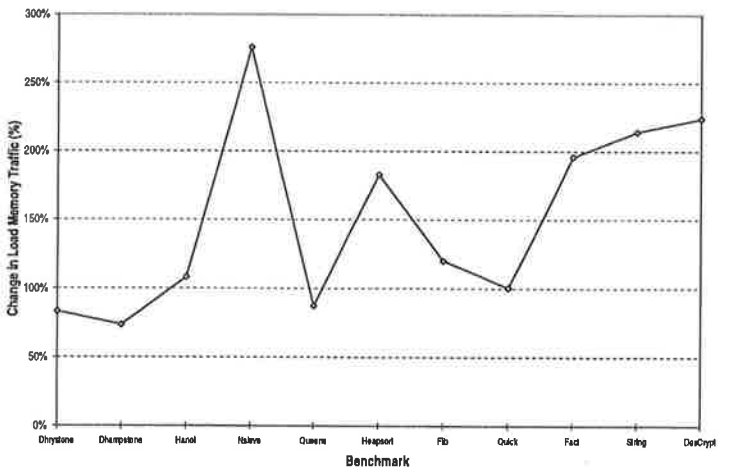
Figure 8.34 Branch Predictor Performance on Benchmarks.



(a) Hit Rate



(b) Load Stall Rate



(c) Relative Load Memory Traffic to DCache (compared to Base Machine)

Figure 8.35 LRB Performance on Benchmarks. Note that, for the benchmarks continuing to perform well using LRBs, hit and load stall rates are mostly *improved* over the idealised scheme of Figures 8.22 and 8.23. This is due to improvements in the prefetching algorithms used.

mance loss was traced to the method the compiler uses for incrementing loop variables in these programs, shown here for the example of HEAPSORT,

```
Load_Values :
   add     r1, ...    ; set memory index register for this loop iteration
   lw      rx,(r1)    ; load first value
   lw      ry,4(r1)   ; load second value
    ...     ; compute on values
```

The LRB contains a cached, valid index for `r1` when the ADD occurs. This entry must be then marked as *Pending·Burned* until the update occurs when the ADD reaches MEM. At this time, the MEM stage receives the `add r1` instruction, but the LRB entry cannot be marked $\overline{Pending} \cdot \overline{Burned}$ because the MEM stage must first determine if the DCache has the data the update is referring to. The ID stage cannot know that the result of the `add r1` instruction will be *necessary* for the arriving loads when it issues the instruction.

However, if the compiler were changed to either post-modify the variable rather than pre-modify, or to move the `add r1` instruction back two instructions (allowing the LRB entry to be valid and updated when the first of the `lw` instructions arrives at ID), then the hit rate for the LRB on this code would jump from 0% to 100%. Problems with extremely poor LRB performance were traced in every case to code of this nature, and this compiler optimisation would be relatively trivial.

This was not uncovered on the preliminary examination because all functionality embodied by the LRB was in the MEM stage, making updating and subsequent hits for arriving loads perfect. Interestingly, all of the poorly performing benchmarks of Figure 8.35 were found to have LRB hit rates of close to 100% and effectively 0% load stalls on the idealised model. This would suggest that this compiler optimisation would be *extremely* effective. The compiler could also be changed to use unsigned operations for changing address values on loads wherever possible (giving a slight speed increase).

Using this compiler optimisation could be expected to improve Amedo performance another 5 to 10%, and significantly reduce memory traffic. In addition, improvements in the LRB control algorithm certainly seem possible given more examination of the dynamic behaviour of memory-referencing instructions, and changes in the compiler to optimise for the LRB structure would assist in generating more powerful algorithms for LRB control.

Note that, on the benchmarks with such problems with the LRB control, the memory traffic, shown in Figure 8.35(c), is *massively* increased, and this is definitely a problem. This is because the LRB is continually missing, and also prefetching many cached registers, which

never hit because of problems in the compiled code. However, even with the moderate hit rates provided by some benchmarks, the load memory traffic is under that of the base machine. Again, compiler optimisations would largely mitigate the traffic increase problem.

3.2 Comparative Performance

The performance of Amedo relative to existing processors is shown in Table 8.5.

Machine	AMULET1	AMULET2e	SPARC^{tm}-5	Base	Amedo	UltraSPARC^{tm}
Tech.	1.0μm	0.5μm	0.5μm	0.8μm	0.8μm	0.45 μm
Clock	—	—	85MHz	66MHz	—	167MHz
Ref.	[WDF+97]	[FGT+97]	[Mic]	—	—	[LCT+95]
Dhryst'ns	20.5k	74k	114k	98k	154k	337k

Table 8.5 Performance Comparisons. The integer *Dhrystone* measure for various processors is compared. SPARC^{tm}-based measurements were based on the compiled benchmark using `cc -fast`. The *Base* machine is an idealised RISC.

Although both SPARC machines tested are multiple-user commercial machines, scaling the clock speed of the base machine to 85MHz gives a 11% difference between the base machine and the SPARC-5, giving one measure of the effect of the idealised memory system. This comparison, while simplified and using only one representative benchmark (based on published figures), shows that Amedo is highly compeditive in terms of performance. The AMULET machines are used as a comparison point as they are the only implemented machines that have a level of functionality enabling the performance of a real benchmark like *Dhrystone* to be evaluated. In fairness, the implementation goals of AMULET are a tradeoff between low power and performance, and the Amedo machine uses *heavily optimised* delay and circuit models, whereas the AMULET machines use speed-independent controllers and a different machine architecture (ARM, as opposed to DLX for Amedo). Furthermore, the AMULET machines are real machines with real memory systems, whereas Amedo is a simulated machine with a partially idealised memory system (for instruction fetch and requests from the DCache), and the timing parameters used are not expected to exactly mirror what will be achieved in implementation.

4 Conclusion

This chapter has presented the Amedo processor, an asynchronous architecture using the FeFA design approach that improves upon the performance of an idealised synchronous base

machine by factors ranging from 52% to 81%.

The characteristics of compiled code executed on a machine virtually identical to early RISC machines (pre-superscalar processors, such as the R2000 [KH92] and SPARC [HB90]) were explored in detail, and an analysis of the dynamic behaviour of both memory accesses and branches was undertaken. The exploration of this base machine was then used to guide the design of Amedo, a free-flow implementation of the DLX architecture. Both the base machine and Amedo have a similar, linear-pipelined architecture. Amedo uses free-flow as the pipeline control mechanism. The datapath architecture and the circuits central to the control of the pipeline were explored. A timing model for both the synchronous base machine and the *Red* timing mechanism used Amedo was developed, using timing parameters gleaned from the implementation of ECSTAC in a 0.8μm CMOS process. This timing model was then added to the execution model of the machine, and both the base and Amedo performance on a range of integer benchmark programs was evaluated. Amedo showed a sustained performance level ranging from 83 to 101 integer MIPS.

Only some features of the Amedo architecture could be potentially integrated with a synchronous implementation. Branch prediction would improve the performance of the synchronous implementation by removing one source of CPI loss. LRBs could be used, but their only use would be elimination of load stalls, since the synchronous model would have to assume that the memory stage time was the full DCache memory access time. In Amedo, LRBs play a central role in improving the MEM stage access time, and the LRB structure allows the determination of a hit ahead of time, making *Red* pipelining using LRBs possible. LRBs also reduce the occurence of one source of performance degradation in the base machine, *load stalls*, by a significant level in the Amedo architecture.

4.1 Related Work

STRiP is a pseudo-synchronous linear pipelined processor [Dea92], described in Chapter 2.2. STRiP used a similar approach to Amedo both in signalling and architectural motivation. The dynamic clocking mechanism used in STRiP exploits gross variations in unit latencies to improve performance, as does Amedo using the *Red* free-flow variable latency method. In addition, free-flow is designed for low overhead in pipelined operation. STRiP also uses a strategy in the MEM stage to improve average-case performance, in this case Level-0 caches. Level-0 caches (in both Fetch and Memory stages) are small, fully-associative buffers with crafted prefetch algorithms that produce a good hit rate in a faster time than that possible with the larger Level-1 ICache and DCache units. A different approach was used in Amedo, but the goal was the same – improve average performance when dealing with memory stages. Amedo uses a two-slot fetch path in the N-Pipe, and Load Register Buffers to improve

average-case performance in the Memory stage. These are totally different solutions to the same common foreseen problem.

4.2 Future Work

No work to date has considered asynchronous pipelining explicitly designed for the performance that free-flow is intended to provide. This has an impact on the architectures that may be employed, as some may not fit with the mechanisms provided by free-flow.

The expansion of the Amedo architecture can progress on a number of fronts. The prefetching and operational algorithms for Load Register Buffers (LRBs) should be explored in more detail, as could similar architectures for the reduction of memory traffic. The insertion of LRBs may change how the DCache should optimally operate. The interaction of the LRB and the DCache should be explored, with a view to incorporating efficient prefetching and access methods in the DCache. The interaction with the compiler should also be addressed, as the success of the LRB algorithm depends on how the compiler uses memory instructions and when it generates the indexes for these memory instructions. Performing compiler optimisations for LRBs should improve their performance considerably.

Widening the execution path of Amedo will be challenging. This will require serious attention to be given to bandwidth issues in the N-Pipe, and fast, efficient slotting and steering mechanisms to be created for moving instructions to the appropriate functional block inputs. Widening the fetch path, improving fetch bandwidth to the execution units, will also have an impact on the design and implementation of the branch prediction engine. Using a separate *memory pipeline*, common practice in superscalar processors [Joh91], will also impact on the architecture and design of Load Register Buffers, and it will be interesting to see if their structure can be adapted to such a system implementation. The integration of floating-point execution with the existing model will be very challenging. The FP unit will be loosely coupled with the integer unit, but interactions will be required. In addition, the lack of synchronisation will make in-order result commitment and maintenance of a defined exception behaviour equally challenging.

Amedo is an interesting asynchronous architecture for the high-speed pipelined execution of integer code. Implementation and further architectural work on Amedo are expected to shed new light on the performance improvement realisable by the free-flow method.

Chapter 9

Conclusion

IMPROVING the performance of asynchronous systems has been the foremost objective of this work. This has been achieved through the use of two methodologies for asynchronous design.

Event Controlled Systems (ECS) is an existing methodology for the specification, design and simulation of two-phase asynchronous systems, primarily using bounded-delay assumptions. ECS brings structure and a method for gate-level description to the design of two-phase asynchronous systems, previously pursued in a more ad-hoc manner. Large networks of two-phase gates can be simulated and checked for timing and signalling errors using definitions readily integrated with commercial tools for hardware description.

Notational difficulties with the ECS approach were eased with the introduction of ECS′, which enables the development of a simple graphical technique for the description of control flows in two-phase circuits. These *Event Flow Graphs* have been used in simplified form throughout this thesis to describe control flows in critical circuits.

The control of pipelined asynchronous systems was explored using the ECS approach. The ECS pipeline developed, the *State Pipeline* or S-Pipe, demonstrated dramatic performance gains over existing approaches in every metric measured — over 50% in both cycle time and merit figures, compared to existing approaches. The D-Pipe (using edge-triggered registers as opposed to latches) and P-Pipe (requiring tighter timing constraints) improve slightly on the cycle time of the S-Pipe, and could be used in performance-critical sections of a design.

ECSTAC, the microprocessor designed using ECS, demonstrated the design task for the two-phase asynchronous design methodology espoused. It shows that fast two-phase control circuits can be built without excessive overhead. The cache control design also identified paths

to improving the performance of control circuits for complex tasks in future implementations. Unfortunately, the performance of ECSTAC is constrained, not by circuits or methodology, but by *architecture*. ECSTAC is a complex device at 220 000 transistors, and the preliminary test results on the fabricated device are positive. Perhaps most importantly, the design showed that the large-scale use of two-phase bounded-delay control was feasible. The project was completed using two designers only over a relatively short time period (the total effort was approximately 16 man-months). ECSTAC was crucial in the development of new ideas using the ECS approach.

The work on ECS up to this point showed that 2ϕ control is fast, efficient, and that using BD control to achieve performance gains is not problematic when timing constraints are properly defined. However, even though ECS pipelines outperform existing methods, there is still a performance bottleneck due to critical path control overhead. This is impossible to overcome in traditional asynchronous pipelines.

Free-Flow, a methodology for asynchronous pipeline control, eliminates the performance bottleneck created by the $req \rightarrow ack$ mechanism in traditional asynchronous pipelines. It relies on a higher degree of timed behaviour in datapath and control circuits, and requires more detailed and specific knowledge on the part of the designers. However, the performance payoff will make this worthwhile — an improvement of 20% over ECS at a logic depth of 8 gates, which will increase as the width of the datapath increases.

Free-Flow does lose a number of aspects considered central to the asynchronous paradigm. Changes in design properties, principally circuit delay, will not be tolerated well by free-flow. There is some ability to absorb parameter variations, particularly with the use of well-considered circuit design techniques for delays, and this should be investigated further. Timing variations due to data dependencies in the computation path cannot be exploited profitably in free-flow, as it is impossible to vary the pipeline operating speed dynamically in response to such unpredictable variations.

The design of delay elements to realise minimum *delay skew* while retaining programmability was explored. Although this is important in ECS, it becomes *critical* in FeFA because of the heavy dependence of system performance on the implementation characteristics of delay elements. One method for the reduction of delay skew by circuit techniques was identified — *Static Timing Skew Compensation* (STSC). The cost of STSC in area and power will be minimal, since the bias generators required can be shared over large portions of a single chip without concern for skew management, as the STSC bias lines are capacitive only.

A scheme for channel communication between asynchronous modules was explored using free-

flow, *Non-Acknowledging Communication* (NAC). NAC demonstrates the tradeoffs between throughput and latency in such channels, and also demonstrated the control of several critical subsystems extensively employed in free-flow.

Using free-flow, an asynchronous microprocessor was developed whose explicit implementation goal is to realise higher performance through the highly efficient pipelining that free-flow provides. The Amedo machine is an exciting architecture for a number of reasons. It may offer above-synchronous performance, depending on how the implementation aspects of the design are managed. It also introduces *Load Register Buffers* (LRBs), which can eliminate many load stalls from the dynamic execution trace of the machine, in addition to offering significant improvements in memory stage load execution time. However, on some benchmarks the performance of LRBs was poor, and the problem causing this was described, and should be simple to correct in a dedicated compiler for the machine. With a simulated sustained integer execution rate ranging from 83 to 101 MIPS in 0.8μm CMOS, Amedo is competitive with existing high-performance asynchronous microprocessors [FGT+97, Mor97].

This thesis has demonstrated that the use of bounded-delay two-phase control realises significant performance gains over existing methods which employ more conservative delay models and design practices. Interestingly, the constraints that ensure the bounded-delay circuits operate correctly have not proved difficult to identify, and often have significant margin that allows a degree of freedom in the implementation style used. This shows that, although bounded-delay methods are traditionally considered risky or not robust, this method of two-phase design is reliable and is a manageable design task.

The use of a two-phase signalling approach also scaled from the initial work on ECS through to the most advanced use of free-flow pipelining, showing that two-phase methods using BD assumptions are durable and expandable across a wide range of performance optimisation methods.

1 The Path Onwards

The continued development of the two asynchronous design methods detailed in this work is essential. ECS is becoming a mature approach for asynchronous design, since the specification mechanism, the notation, and the gates used are well-understood. Expanding ECS will require further work on analysis and CAD tools (the use of representations similar to EFGs will assist in this effort). CAD tools for critical path analysis, optimisation, and timing analysis (both at the gate level and using back-end verification) are essential if the load on

designers is to be reduced in future implementations using the ECS approach.

There is much to be done to bring Free-Flow to this level of usability. Free-Flow demands a more detailed approach to design, requiring more techniques, tools and examples of design to be developed. Although this thesis has developed some fundamental architectures for free-flow systems (pipeline halt, variable latency, and multi-rate), there is much that could be done. System composition is still a large task, and finding methods for the efficient composition of system components is required. Such methods include better techniques for the control of separate pipelines, dataflow within an individual pipeline (data forwarding and backwarding mechanisms being two examples of this), and generalised pipeline interactions.

Free-Flow also places much greater emphasis on the design of *delays*. Low-skew delays are essential in free-flow — any skew in delays impacts directly on system cycle time. Since the STSC technique for skew compensation removes almost all skew by design, confidence in the simulation results should be improved by *testing* STSC and associated techniques for delay skew minimisation through fabrication. In this way, the performance of the delays and the minimisation techniques can be examined in a practical context. Furthermore, alternative methods for delay skew minimisation through circuit techniques should be explored to improve upon the skew performance of the STSC scheme. In addition, CAD tools which automate the design and simulation-based evaluation of delay elements would be very useful, removing the requirement for *ad-hoc* design of delay elements by the designer (which would also aid ECS implementations).

The Amedo machine, designed explicitly for FeFA implementation, should be built. This will allow the exploration of a great many new areas in architecture, control, and circuit design for free-flow. In addition, the demonstration of an above-synchronous performance machine *in practice* would be a significant win for this approach. The Amedo machine still needs more exploration before full implementation commences. The compiler and its interaction both with the basic Amedo architecture, but especially the LRB unit, needs to be addressed in order to maximise the potential performance of the machine. The LRB algorithm and control structure can be developed further to improve hit rate and reduce memory traffic. In addition, it may be possible to use NAC-style communication schemes on the chip to save wiring resources, and the impact on system performance of such additions at various points (for example, the line fill path from the two caches to the memory interface *E-Box*) should be evaluated. These aspects considered together, in addition to the potential performance level of the architecture, make for exciting times ahead!

It is clear that the use of BD engineered approaches holds tremendous potential for improving the performance of asynchronous designs. This has been demonstrated through the

plethora of controllers in this thesis which dramatically improve on the performance of SI controllers, with simple timing verification requirements. The further development of the free-flow methodology is expected to realise the performance improvements detailed in this thesis, using a combination of architecture, BD timing constraints, and powerful circuit techniques.

Publications

Shannon V. Morton, Sam S. Appleton, Michael J. Liebelt,
"An Event-Controlled Reconfigurable Multi-Chip FFT",
International Symposium on Advanced Research in Asynchronous Circuits & Systems,
Salt Lake City, Utah, November 1994.

Shannon V. Morton, Sam S. Appleton, Michael J. Liebelt,
"ECSTAC : A Fast Asynchronous Microprocessor",
2nd Working Conference on Asychronous Design Methodologies,
SouthBank University, London, May 1995.

Sam S. Appleton, Shannon V. Morton, Michael J. Liebelt,
"The Design of a Fast Asynchronous Microprocessor",
IEEE Technical Committee on Computer Architecture Newsletter,
October, 1995.

Sam S. Appleton, Shannon V. Morton, Michael J. Liebelt,
"Cache Design for an Asynchronous VLSI RISC processor",
13th Australian Microelectronics Conference,
Adelaide, Australia, July 1996.

C. Farnsworth, P. Day, D.A. Edwards, Shannon V. Morton, Sam S. Appleton,
Michael J. Liebelt,
"Asynchronous Pipelining Techniques and Applications",
13th Australian Microelectronics Conference,
Adelaide, Australia, July 1996.

Sam S. Appleton, Shannon V. Morton, Michael J. Liebelt,
"High-Performance Two-Phase Asynchronous Pipelines",
IEICE ED Journal,
March 1997.

Sam S. Appleton, Shannon V. Morton, Michael J. Liebelt,
"A New Technique for Asynchronous Pipeline Control"†,
IEE Electronics Letters,
October 1996.

Sam S. Appleton, Shannon V. Morton, Michael J. Liebelt,
"A New Method for Asynchronous Pipeline Control"†,
7th Great Lakes Symposium on VLSI,
March 1997, Urbana, Illinois, USA.

Sam S. Appleton, Shannon V. Morton, Michael J. Liebelt,
"Two Phase Asynchronous Pipeline Control",
Third International Symposium on Advanced Research in Asynchronous Circuits and Systems,
April 1997, Eindhoven, Netherlands.

†These papers use the previous name for FeFA, being FOCA (*Flow Controlled Asynchronism*). The two names are exactly equivalent in terms of what is described, and simply reflect a change in the thinking processes of the author.

Appendix A

Delay Design

DELAY ELEMENTS have been used extensively throughout this thesis as a critical component of many circuits. Good delay design is central to the free-flow methodology for asynchronous design. This appendix briefly explores some aspects of the performance of simple delay elements.

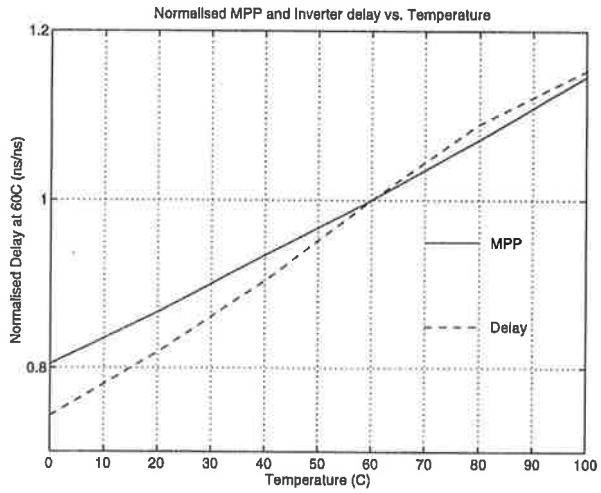
1 Delay Performance and Self-Timed Units

The delay characterisation of Chapter 6 focused on the *skew* of the delay elements, since this has a direct impact on the peak issuing rate. However, the performance of the delay *relative* to what it is modelling (the datapath) is also important.

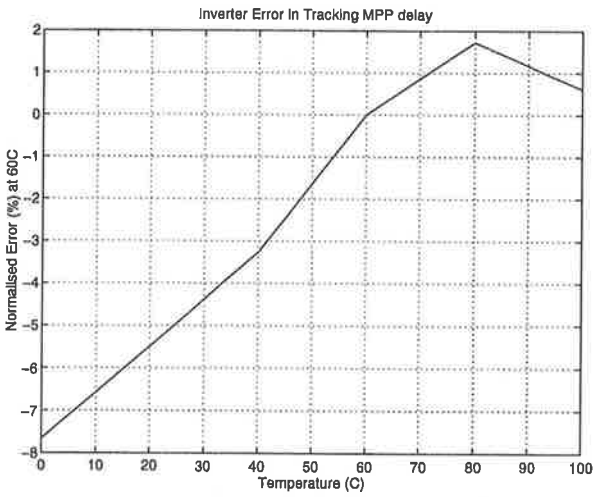
The delay characteristics of a self-timed unit, the MPP [AML97c], and a simple delay element consisting of a short inverter chain were investigated to determine how well the delay element tracked the delay of a self-timed unit over temperature and voltage variations. The results of the HSPICE simulations using a 0.8μm CMOS process [PP94] are shown in Figures A.1 and A.2.

The MPP is a totally autonomous unit with no delay models, and represents roughly how on-chip units should behave in practice. The delay element is a simple inverter chain. This has a number of drawbacks, in particular, the fanout of the circuit is very low, and may not effectively mirror the fanout of a real circuit.

This experiment shows that a delay element tracks the timing of the MPP to within a 5% error over a reasonable range of parameter variation. This has a number of implications on

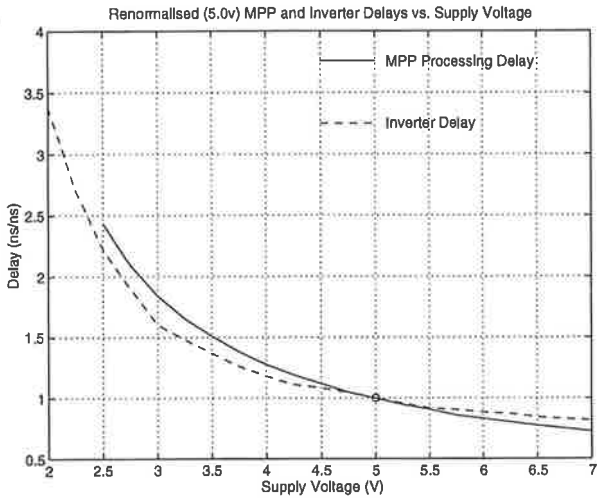


(a) Normalised (5V) Values

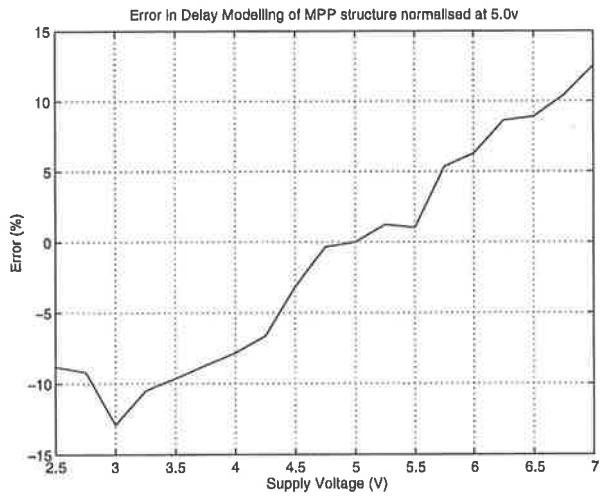


(b) Error

Figure A.1 Delay Element and MPP latencies against Temperature.



(a) Normalised (5V) Values



(b) Error

Figure A.2 Delay Element and MPP latencies against Voltage.

the design of delay elements for use in tracking the real latencies of units.

The delay element can be expected to give a negative error of *at most* 5% for performance over voltage and temperature conditions. Therefore, the nominal value of the delay will have to be overdesigned by approximately 10% to allow for both of these variations. The delay moving into positive error means that the circuit is *over-estimating* the delay of the datapath. This will correpsond to a performance degradation relative to the current overestimation. In addition, if the delay is *over-estimated* nominally, then a greater performance impact will be felt.

This is clearly going to be unacceptable for free-flow, since this will detract from the peak achievable performance of the asynchronous pipeline. Further work should examine the circuit design of delays so that this level of delay margining does not need to be used.

2 Loading Effects

The effect of various design parameters on the delay of a nominally-designed delay element was then tested. The input signal edge rate and output capacitance of the delay element under test were varied, and the results are shown in Figures A.3 and A.4.

Figure A.3 shows that there is a roughly linear relationship between the delay of the element and the input edge rate triggering the delay. However, there appears to be no correlation between the input and output edge rates.

Figure A.4 shows that, as expected, there is a linear relationship between both element delay and output edge rate against output load (capacitance). This means the delay and output edge rate of the element can be characterised by three constants (for each delay element type),

$$\begin{aligned} T_{delay} &= \kappa_{d\,edge} \cdot \Delta T_{i/pedge} + T_{intrinsic} + \kappa_{d\,load} \cdot C_{out} \\ \Delta T_{edge\,out} &= \kappa_{edge\,load} \cdot C_{out} + \Delta T_{edge\,intrinsic} \end{aligned}$$

$\kappa_{d\,edge}$ represents the dependence between input edge rate and element delay, and $\kappa_{d\,load}$ represents the dependence on element delay on the output load. $\kappa_{edge\,load}$ represents the dependence on output edge rate with output loading. All three are simple constants, as the characteristics shown in Figures A.3 and A.4 are approximately linear.

As this is a very simple relation between for the delay of an element based on its input and output environment, it can be applied future work for the development of CAD tools and methodologies for delay element design.

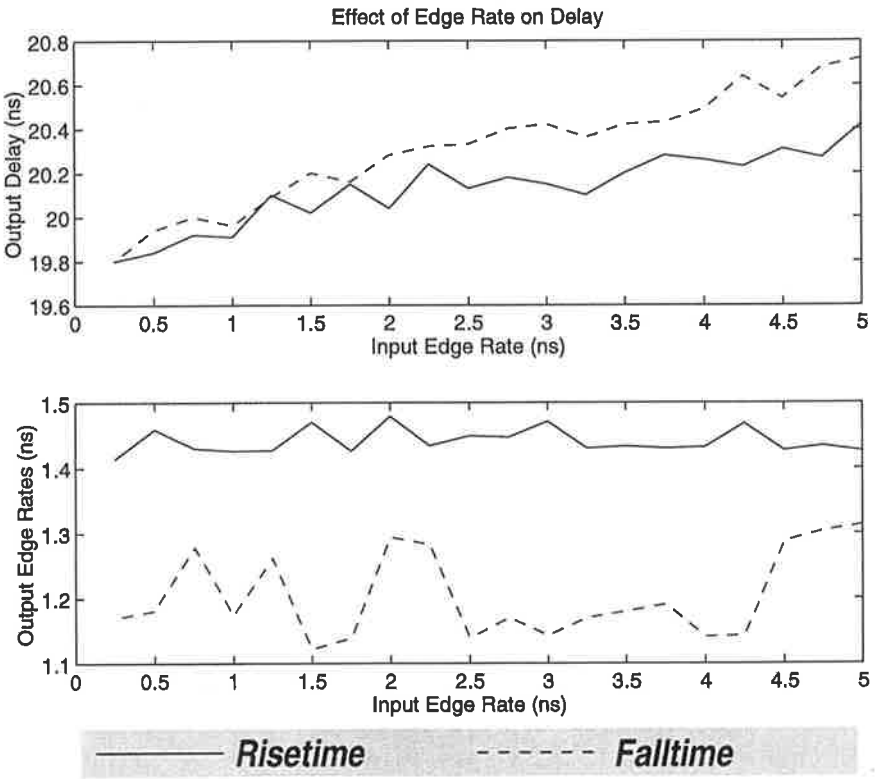


Figure A.3 Input Rate Effects.

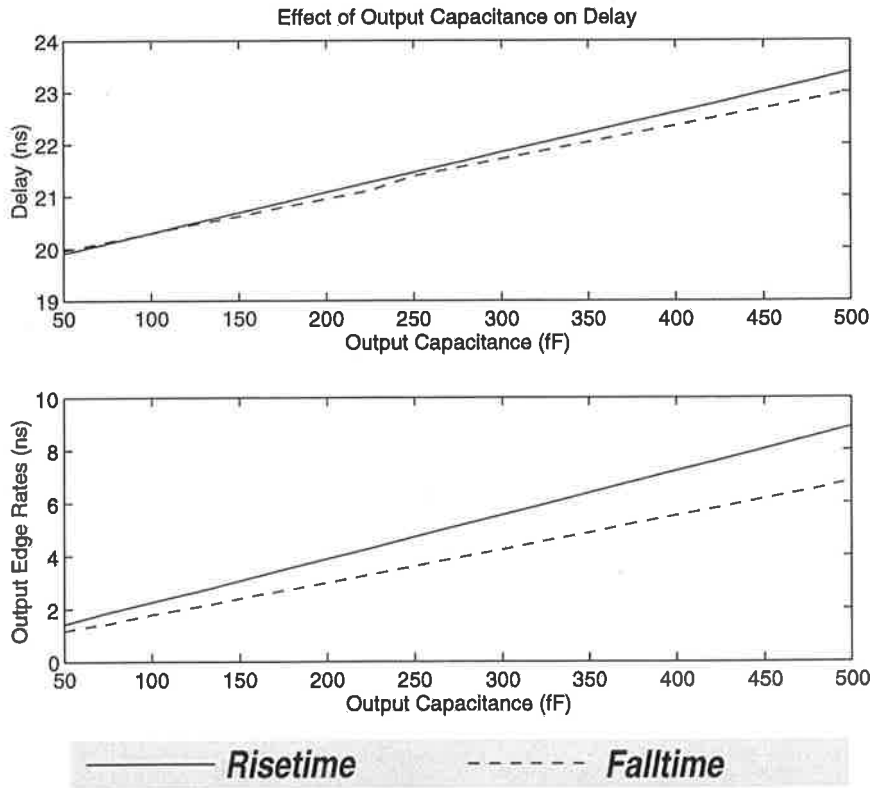


Figure A.4 Output Loading Effects.

Appendix B

Amedo ISA

THE DLX Instruction Set Architecture [PH96,SK96a] is used on Amedo almost without modification. The DLX ISA defines one branch delay slot instruction – the immediately succeeding instruction any control transfer instruction (taken or not) is always executed. In addition, the software does not enforce the load delay slot (required when an instruction used the result of a load instruction that occured on the immediately preeceding cycle), and the hardware must enforce the load interlock dynamically.

The instruction set for Amedo is described in Tables B.1 and B.2.

A range of *(op)* types are available for sets, covering every possible relation between the two operands. Logical operands can be AND, OR and XOR.

Control transfer and miscellaneous instructions are shown in Table B.2.

1 Register Transfer Operation

The register transfer operation of the ID and EX stages of Amedo, without considering forwarding (including the LRB controls) is shown in Tables B.3 and 1, for the instructions in Tables B.1 and B.2, respectively. These RTL mechanisms are for the Amedo ID stage and EX stage, shown in Figures 8.14 and 8.15. Note that instructions which interact with the user/supervisor status of the machine, for example TRAP and RFE, are not detailed here. The handling of these instructions is dependent on how the machine would be implemented in practice.

Instruction	R -type	I-type	Comment
Add	ADD	ADDI	
Add (Unsigned)	ADDU	ADDUI	
Sub	inssub	SUBI	
Sub (Unsigned)	SUBU	SUBUI	
Shift Left	SLL	SLLI	
Shift Right	SRL	SRLI	
Shift Right Arithmetic	SRA	SRAI	
Set	S(CODE)	S(CODE)	
Set Unsigned	S(CODE)U	S(CODE)UI	
Load High		LHI	Loads High 16-bits of DestReg
Logical	(OP)	(OP)I	
NOR	NOR		only R-type
Load Word		LW	
Load Half/Byte		LH/LB	
Load Half/Byte Unsigned		LHU/LBU	
Store Word		SW	
Store Half/Byte		SH/SB	

Table B.1 Arithmetic and Memory Instructions.

Instruction	I-type	J-type	Comment
Branch on Zero	BEQZ		Branch to PC+4+imm(words)
Branch on not Zero	BNEZ		(same)
Jump		J	Jump to PC+4+imm(words)
Jump and Link		JAL	J & also Link to r31
Jump Register	JR		Jump to Specified Reg. Target
Jump register and Link	JR		JR & also Link to r31
System Trap		TRAP (no.)	

Table B.2 Control Transfer and Miscellaneous Instructions.

Instn.	ID actions	EX actions
Add/Sub/Logical	ID-B ← rs2Val(R)	EXalOut ← EXAop (op) EXBop
	ID-B ← immX(I)	EXresult ← EXalOut
Shifts	ID-B ← rs2Val(R)	EXshOut ← EXAop (op) EXBop
	ID-B ← immX(I)	EXresult ← EXshOut
Sets	ID-B ← rs2Val(R)	EXalOut ← EXAop - EXBop
	ID-B←immX(I)	EXsetOut ← $\mathcal{F}(EXalZero, EXsetType)$
		EXresult ← EXsetOut
Loads	ID-B ← immX	EXalOut ← EXAop + EXBop
Stores	ID-S ← rs2Val	EXalOut ← EXAop + EXBop
	ID-B ← immX	EXspecOut ← EXspecialValue

Table B.3 Amedo RTL description for Integer and Memory Instructions.

Instn.	ID actions	EX actions
J	DIUTR	SquashEX (no result)
JAL	DIUTR	EXLinkTarget ← EXPC+8
		EXresult ← EXLinkTarget
JR	DIUTR	EXalOut ← EXAop ⊕EXBop
	ID-B←ID-PPC	if (EXalOut ≠ 0)
		newPCfromX ← EXAop
		assert *XPipeMispredict*
		Run NPC Reload Sequence
		Squash EX (no result)
JALR	DIUTR	EXalOut ← EXAop ⊕ EXBop
	ID-B←ID-PPC	EXspecialValue ← PC+8
		EXresult ← EXspecialValue
		if (EXalOut ≠ 0)
		newPCfromX ← EXAop
		assert *XPipeMispredict*
		run NPC reload sequence
BNEZ/BEQZ	DIUTR	if (mispredict)
		newPCfromX ← EX
		newPCfromX ← EXspecialValue (taken)
		newPCfromX ← PC+8 (not taken)
		assert *XPipeMispredict*
		run NPC reload sequence
		Squash EX (no result)

Table B.4 Amedo RTL description for Control Transfers. DIUTR stands for *Disable Interrupts Until Target Reached.*

Appendix C

Gate Delay Data

THE use of *gate delays* (or $\mathcal{T}_g$'s) in this thesis is based on a characterisation of a $0.8\mu m$ CMOS process and the designed library. This library was custom designed and characterised, and used in the implementation of ECSTAC (see Chapter 5).

Gate delay data are summarised in Table C.1. Absolute gate delays have a 25fF load representing an equivalent gate load of one, except where otherwise noted.

Gate	Delay/Equiv. Gate Load	Gate Delay ($\mathcal{T}_g$)
Send	1.3ns/1	1.5
Until/Merge (XOR)	600ps/1	0.7
Until-Loaded	700p/8	0.8
	1.0ns/32	1.2
	1.3ns/64	1.5
Last	1.6ns/1	1.8
Latch (inverting)	1.3ns/1	1.5
Latch (dynamic, non-inverting)	910ps/1	1.1
nand(2)	300ps/1	0.35
nor(2)	350ps/1	0.40

Table C.1 Relevant Gate Delay Values for 0.8μm CMOS technology.

Bibliography

[ABF90] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design.* IEEE Press, Revised edition, 1990.

[Adv95a] Advanced Microelectronics Inc. *IRSIM Users Manual*, 1995.

[Adv95b] Advanced Microelectronics Inc. *MAGIC Users Manual*, 1995.

[AM93] Sam S. Appleton and Shannon V. Morton. Design of a controller for an Event Controlled Cache System. Report HPCA-ECS-93/07, Dept. of Electrical and Electronic Engineering, The University of Adelaide, Adelaide, South Australia, June 1993.

[AMJL94] Sam S. Appleton, Shannon V. Morton, Andrew B. Johnson, and Michael J. Liebelt. ECSTACBus: External Bus Protocol for the ECSTAC-P Microprocessor. Report HPCA-ECS-94/3, Dept. of Electrical and Electronic Engineering, The University of Adelaide, Adelaide, South Australia, 1994.

[AML94a] Sam S. Appleton, Shannon V. Morton, and Michael J. Liebelt. Instruction and Data Caches for the ECSTAC Microprocessor. Report HPCA-ECS-94/7, Dept. of Electrical and Electronic Engineering, The University of Adelaide, Adelaide, South Australia, 1994.

[AML94b] Sam S. Appleton, Shannon V. Morton, and Michael J. Liebelt. Metastability in ECS Systems. Report HPCA-ECS-94/8, Dept. of Electrical and Electronic Engineering, The University of Adelaide, Adelaide, South Australia, 1994.

[AML95] Sam S. Appleton, Shannon V. Morton, and Michael J. Liebelt. Instruction and Data Caches for the ECSTAC-P Microprocessor. Report HPCA-ECS-95/04, Dept. of Electrical and Electronic Engineering, The University of Adelaide, Adelaide, South Australia, February 1995.

[AML96] Sam S. Appleton, Shannon V. Morton, and Michael J. Liebelt. Technique for High Speed Asynchronous Pipeline Control. *Electronics Letters*, 32(21):1973–1974, October 1996.

[AML97a] Sam S. Appleton, Shannon V. Morton, and Michael J. Liebelt. A New Method for Asynchronous Pipeline Control. In *7th Great Lakes Symposium on VLSI*, March 1997.

[AML97b] Sam S. Appleton, Shannon V. Morton, and Michael J. Liebelt. High Performance Two-Phase Asynchronous Pipelines. *IEICE ED Transactions*, March 1997.

[AML97c] Sam S. Appleton, Shannon V. Morton, and Michael J. Liebelt. Two-Phase Asynchronous Pipeline Control. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, April 1997.

[AMU96] *AMULET2e Datasheet – 32-bit Integrated Asynchronous Microprocessor*, 1996. Web Reference *www.cs.man.ac.uk/amulet/AMULET2e_uP.html*.

[App96] Sam Appleton. Implementation of Instruction and Data Caches for the ECSTAC Microprocessor. Report HPCA-ECS-96/02, Dept. of Electrical and Electronic Engineering, The University of Adelaide, Adelaide, South Australia, June 1996.

[AY91] M. Afghahi and J. Yuan. Double Edge-Triggered D-Flip-Flops for High-Speed CMOS Circuits. *IEEE Journal of Solid-State Circuits*, 26(8), August 1991.

[Bak90] H.B. Bakoglu. *Circuits, Interconnections and Packaging for VLSI*. Addison-Wesley, 1990.

[BB96] Kees van Berkel and Arjan Bink. Single-track handshaking signaling with application to micropipelines and handshake circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 122–133. IEEE Computer Society Press, March 1996.

[BBB+95] B.J. Benschneider, A.J. Black, W.J. Bowhill, S.M. Britton, D.E. Dever, D.R. Donchin, R.J. Dupcak, R.M. Fromm, M.K. Gowan, P.E. Gronowski, M. Kantrowitz, M.E. Lamere, S. Mehta, J.E. Meyer, R.O. Mueller, A. Olesin, R.P. Preston, D.A. Priore, S. Santhanam, M.J. Smith, and G.M. Wolrich. A 300-MHz 64-b Quad-Issue CMOS RISC Microprocessor. *IEEE Journal of Solid-State Circuits*, 30(11), November 1995.

[BBK+94a] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, and Frits Schalij. A Fully-Asynchronous Low-Power Error Corrector for the DCC Player. In *International Solid State Circuits Conference*, pages 88–89, February 1994.

[BBK+94b] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, and Frits Schalij. Asynchronous Circuits for Low Power: A DCC Error Corrector. *IEEE Design & Test of Computers*, 11(2):22–32, Summer 1994.

[BBK+94c] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, and Frits Schalij. A Fully-Asynchronous Low-Power Error Corrector for the DCC Player. *IEEE Journal of Solid-State Circuits*, 29(12):1429–1439, December 1994.

[BBK+95] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, Frits Schalij, and Rik van de Wiel. A Single-Rail Re-implementation of a DCC Error Detector Using a Generic Standard-Cell Library. In *Asynchronous Design Methodologies*, pages 72–79. IEEE Computer Society Press, May 1995.

[Ber92a] Kees van Berkel. Beware the Isochronic Fork. *Integration, the VLSI journal*, 13(2):103–128, June 1992.

[Ber92b] Kees van Berkel. *Handshake Circuits: An Intermediary between Communicating Processes and VLSI.* PhD thesis, Eindhoven University of Technology, 1992.

[Bou96] Gregg Bouchard. Design Objectives of the 0.35μm Alpha 21164 Microprocessor. In *HotChips 8*, August 1996.

[Bra93] Brain K. Bray. *Specialsed Caches to Improve Data Access Performance.* PhD thesis, Department of Electrical Engineering, Stanford University, May 1993.

[Bre75] Melvin A. Breuer, editor. *Digital System Design Automation : Languages, Simulation & Database.* Digital System Design Series. Computer Science Press, 1975.

[Bru91] Erik Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits.* PhD thesis, Carnegie Mellon University, 1991.

[BS95a] Andrew Beaumont-Smith. Gallium Arsenide Design Methodology and Testing of a Systolic Floating Point Processing Element. Master's thesis, Department of Electrical and Electronic Engineering, University of Adelaide, 1995.

[BS95b] Janusz A. Brzozowski and Carl-Johan H. Seger. *Asynchronous Circuits.* Springer-Verlag, 1995.

[Bur] Tom Burd. The CPU Info Center. Web Reference *infopad.eecs.berkeley.edu/CIC.*

[Bur88] Steven M. Burns. Automated Compilation of Concurrent Programs into Self-timed Circuits. Master's thesis, California Institute of Technology, 1988.

[Bur91] Steven M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits.* PhD thesis, California Institute of Technology, 1991.

[CCS+88] Babara A. Chappell, Terry I. Chappell, Stanley E. Schuster, Hermann M. Segmuller, James W. Allan, Robert L. Franch, and Phillip J. Restle. Fast CMOS ECL Receivers with 100-mV Worst-Case Sensitivity. *IEEE Journal of Solid-State Circuits*, 23(1), February 1988.

[Cho89] Paul Chow, editor. *The MIPS-X RISC Microprocessor.* Kluwer Academic Publishers, 1989.

[Chu87] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications.* PhD thesis, MIT Laboratory for Computer Science, June 1987.

[CL95] Chih-Ming Chang and Shih-Lien Lu. Design of a Static MIMD Data Flow Processor Using Micropipelines. *IEEE Transactions on VLSI Systems*, 3(3):370–378, September 1995.

[CLM94] Uri Cummings, Andrew Lines, and Alain Martin. An Asynchronous Pipelined Lattice Structure Filter. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 126–133, November 1994.

[CM73] T. J. Chaney and C. E. Molnar. Anomalous Behavior of Synchronizer and Arbiter Circuits. *IEEE Transactions on Computers*, C-22(4):421–422, April 1973.

[Coo93] James N. Cook. Production Rule Verification for Quasi-Delay-Insensitive Circuits. Master's thesis, California Institute of Technology, June 1993.

[Cor93] Henk Corporaal. Evaluating *Transport Triggered Architectures* for scalar applications. Technical report, Delft University of Technology, Netherlands, 1993. Web Reference : einstein.et.tudelft.nl/~heco/documents/documents.html.

[Cor97] Standard Performance Evaluation Corporation. *SPEC CPU95*, May 1997. Version 1.10.

[Cot65] Leonard W. Cotten. Circuit Implementation of High-Speed Pipeline Systems. In *AFIPS Proceedings Fall Joint Computer Conference*, volume 27, pages 489–504, 1965.

[Cot69] Leonard W. Cotten. Maximum-rate Pipeline Systems. In *AFIPS Proceedings Spring Joint Computer Conference*, volume 34, pages 581–586, 1969.

[DAC+92] Daniel W. Dopperpuhl, Robert Anglin, Linda Chao, Bruce Gieseke, Kathryn Kuchler, Liam Madden, James Montanaro, Sridhar Samudrala, Richard T. Witekn, Davoid Bertucci, Robert A. Conrad, Soha M.N. Hassoun, Maureen Ladd, Edward J. McLellan, Donald A. Priore, Sribalan Santhanam, Randy Allmon, Sharon Britton, Daniel E. Dever, Gregory W. Hoeppner, Burton M. Leary, Derrick R. Meyer, and Vidya Rajagopalan. A 200-MHZ 64-bit Dual-Issue CMOS Microprocessor. *Digital Technical Journal*, 4(4), 1992.

[DDH94] Mark E. Dean, David L. Dill, and Mark Horowitz. Self-Timed Logic Using Current-Sensing Completion Detection (CSCD). *Journal of VLSI Signal Processing*, 7(1/2):7–16, February 1994.

[Dea92] Mark E. Dean. *STRiP: A Self-Timed RISC Processor Architecture.* PhD thesis, Stanford University, 1992.

[Dij75] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8), August 1975.

[Dil89] David L. Dill. *Trace Theory for Automatic Hierachical Verification of Speed-Independent Circuits.* ACM Distinguished Dissertations. MIT Press, 1989.

[DS91] Marios D. Dikaiakos and Kenneth Steiglitz. Comparison of Tree and Straight-Line Clocking for Long Systoloc Arrays. *Journal of VLSI Signal Processing*, pages 1177–1180, 1991.

[DW95] Paul Day and J. Viv Woods. Investigation into Micropipeline Latch Design Styles. *IEEE Transactions on VLSI Systems*, 3(2):264–272, June 1995.

[EB95] Jo Ebergen and Robert Berks. VERDECT: A Verifier for Asynchronous Circuits. *IEEE Technical Committee on Computer Architecture Newsletter*, October 1995.

[Ebe87] J.C. Ebergen. *Translating Programs into Delay Insensitive Circuits.* PhD thesis, Technical University of Eindhoven, 1987.

[ECFS95] C. J. Elston, D. B. Christianson, P. A. Findlay, and G. B. Steven. Hades - Towards the Design of an Asynchronous Superscalar Processor. In *Asynchronous Design Methodologies*, pages 200–209. IEEE Computer Society Press, May 1995.

[Ell96] Gary Ellis. A Summary of the Clock Optimisation Problem. Technical report, Department of Electrical and Computer Engineering,Carnegie Mellon University, Pittsburgh, PA, 1996.

[Ell97] Gary Ellis. *The Physical Design of Clock Circuits.* PhD thesis, Carnegie Mellon University, April 1997.

[End95] Phillip B. Endecott. *SCALP : A Superscalar Asynchronous Low-Power Processor.* PhD thesis, University of Manchester, 1995.

[End97a] P. Endecott. LARD CAD Tool Demo. In *Third International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 1997. CAD Tool Demonstration.

[End97b] P. Endecott. *LARD Language Reference Manual*, 1997. Web Reference *www.cs.man.ac.uk/amulet/projects/lard/langref/index.html.*

[EPR97] Gary Ellis, Lawrence T. Pileggi, and Rob A. Rutenbar. A Hierachial Decomposition Methodology for Single-Stage Clock Circuits. In *Proc. IEEE CICC*, 1997.

[ESB95] Jo C. Ebergen, John Segers, and Igor Benko. Parallel Program and Asynchronous Circuit Design. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 51–103. Springer-Verlag, 1995.

[FD96] Stephen B. Furber and Paul Day. Four-Phase Micropipeline Latch Control Circuits. *IEEE Transactions on VLSI Systems*, 4(2):247–253, June 1996.

[FDG+94] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. AMULET1: A micropipelined ARM. In *Proceedings IEEE Computer Conference (COMPCON)*, pages 476–485, March 1994.

[FGT+97] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N.C. Paver. AMULET2e: An Asynchronous Embedded Controller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems.* IEEE Computer Society Press, April 1997.

[FK85] Allan L. Fisher and H.T. Kung. Synchronising Large VLSI Processor Arrays. *IEEE Transactions on Computers*, C-34(8):734–740, August 1985.

[FL96] S. B. Furber and J. Liu. Dynamic Logic in Four-Phase Micropipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems.* IEEE Computer Society Press, March 1996.

[Fri95] Eby G. Friedman. *Clock Distribution Networks in VLSI Circuits & Systems.* IEEE Press, 1995.

[Fur95] S. Furber. Computing without Clocks: Micropipelining the ARM Processor. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 211–262. Springer-Verlag, 1995.

[Gar93] Jim D. Garside. A CMOS VLSI Implementation of an Asynchronous ALU. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 181–207. Elsevier Science Publishers, 1993.

[GBD+96] P.E. Gronowski, W.J. Bowhill, D.R. Donchin, R.P. Blake-Campos, D.A. Carlson, E.R. Equi, B.J. Loughlin, S. Mehta, R.O. Mueller, A. Olesin, D.J.W. Noorlag, and R.P. Preston. A 433-MHz 64-b Quad-Issue RISC Microprocessor. *IEEE Journal of Solid-State Circuits*, 31(11), November 1996.

[GJ90] Ganesh Gopalakrishnan and Prabhat Jain. Some Recent Asynchronous System Design Methodologies. Technical Report UUCS-TR-90-016, Dept. of Computer Science, Univ. of Utah, October 1990.

[GJ95] E. Grass and S. Jones. Asynchronous Circuits Based On Multiple Localised Current-Sensing Completion Detection. In *Asynchronous Design Methodologies*, pages 170–177. IEEE Computer Society Press, May 1995.

[GLI94] C. Thomas Gray, Wentai Liu, and Ralph K. Calvin III. *Wave Pipelining : Theory and CMOS Implementation.* Kluwer Academic Publishers, 1994.

[GMK96] E. Grass, R. C. S. Morling, and I. Kale. 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.* IEEE Computer Society Press, March 1996.

[GO95] Alessandro De Gloria and Mauro Olivieri. Efficient Semicustom Micropipeline Design. *IEEE Transactions on VLSI Systems*, 3(3):464–469, September 1995.

[Gra93] Carl Thomas Gray. *Optimal Clocking of Wave Pipelined Systems and CMOS Applications.* PhD thesis, Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC, 1993.

[Gre93] Mark Russell Greenstreet. *STARI : A Technique for High-Bandwidth Communication.* PhD thesis, Department of Computer Science, Princeton University, 1993.

[Hau95] Scott Hauck. Asynchronous Design Methodologies: An Overview. *Proceedings of the IEEE*, 83(1), January 1995.

[Haz92] Pieter J. Hazewindus. *Testing Delay-Insensitive Circuits.* PhD thesis, California Institute of Technology, 1992.

[HB90] M. Hall and J. Barry, editors. *The Sun Technology Papers.* Springer-Verlag, 1990.

[Hen84] John Hennessy. VLSI Processor Architecture. *IEEE Transactions on Computers*, 33(12), December 1984.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Huf64] D. A. Huffman. The Synthesis of Sequential Switching Circuits. In E. F. Moore, editor, *Sequential Machines: Selected Papers.* Addison-Wesley, 1964.

[Hul95] Henrik Hulgaard. *Timing Analysis and Verification of Timed Asynchronous Circuits.* PhD thesis, Department of Computer Science, University of Washington, 1995.

[Inm88] Inmos Limited. *OCCAM 2 reference manual*, 1988.

[JK88] Thomas K. Knight Jr. and Alexander Krymm. A Self-Terminating Low-Voltage Swing CMOS Output Driver. *IEEE Journal of Solid-State Circuits*, 23(2), April 1988.

[Joh91] Mike Johnson. *Superscalar Microprocessor Design.* Prentice-Hall, 1991.

[Jou90] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. Technical Note TN-14, Digital Western Research Laboratory, Palo Alto, CA, March 1990. Web Reference *www.research.digital.com/wrl/home.html.*

[JU90a] Mark B. Josephs and Jan Tijmen Udding. Delay-Insensitive Circuits: An Algebraic Approach to their Design. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90, Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 342–366. Springer-Verlag, August 1990.

[JU90b] Mark B. Josephs and Jan Tijmen Udding. The Design of a Delay-Insensitive Stack. In G. Jones and M. Sheeran, editors, *Designing Correct Circuits*, pages 132–152. Springer-Verlag, 1990.

[JU93a] M. B. Josephs and J. T. Udding. An Overview of DI Algebra. In *Proc. Hawaii International Conf. System Sciences*, volume I. IEEE Computer Society Press, January 1993.

[JU93b] Mark B. Josephs and Jan Tijmen Udding. Implementing a Stack as a Delay-Insensitive Circuit. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 123–135. Elsevier Science Publishers, 1993.

[JW89] Norman P. Jouppi and David W. Wall. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines. Research Report 89/7, Digital Western Research Laboratory, Palo Alto, CA, July 1989. Web Reference *www.research.digital.com/wrl/home.html.*

[KdSRA91] S. Karthik, I. de Souza, J. T. Rahmeh, and J. A. Abraham. Interlock Schemes for Micropipelines: Application to a Self-Timed Rebound Sorter. In *Proc. International Conf. Computer Design (ICCD)*, pages 393–396. IEEE Computer Society Press, 1991.

[Kel95] Robert Kelly. Asynchronous Design Aspects of High-Performance Logic: Architectural Modelling of a Bipolar Asynchronous Microprocessor. Master's thesis, Deparment of Computer Science, University of Manchester, 1995.

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture.* Prentice-Hall, 1992.

[KKTV92] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. On Self-Timed Behavior Verification. In *Proceedings of ACM TAU 92*, March 1992.

[KKTV94] Michael Kishinevsky, Alex Kondratyev, Alexander Taubin, and Victor Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design.* Series in Parallel Computing. John Wiley & Sons, 1994.

[Kog81] P.M. Kogge. *The Architecture of Pipelined Computers.* McGraw-Hill, 1981.

[KTT+88] Shinji Komori, Hidehiro Takata, Toshiyuki Tamura, Fumiyasu Asai, Takio Ohno, Osamu Tomisawa, Tetsuo Yamasaki, Kenji Shima, Katsuhiko Asada, and Hiroaki Terada. An Elastic Pipeline Mechanism by Self-Timed Circuits. *IEEE Journal of Solid-State Circuits*, 23(1):111–117, February 1988.

[KTT+89] Shinji Komori, Hidehiro Takata, Toshiyuki Tamura, Fumiyasu Asai, Takio Ohno, Osamu Tomisawa, Tetsuo Yamasaki, Kenji Shima, Hiroaki Nishikawa, and Hiroaki Terada. A 40-MFLOPS 32-bit Floating-Point Processor with Elastic Pipeline Scheme. *IEEE Journal of Solid-State Circuits*, 24(5):1341–1347, October 1989.

[Kun88] S.Y. Kung. *VLSI Array Processors.* Prentice-Hall, 1988.

[LB90] Stephen I. Long and Steven E. Butner. *Gallium Arsenide Digital Integrated Circuit Design.* McGraw-Hill, 1990.

[LCT+95] Lavi A. Lev, Andy Charnas, Marc Tremblay, Alexander R. Dalal, Bruce A. Frederick, Chakra R. Srivatsa, David Greenhill, Dennis L. Wendell, Duy Dinh Pham, Eric Anderson, Hemraj K. Hingarh, Inayat Razzack, James M. Kaku, Ken Shin, Marc E. Levitt, Michael Allen, Phillip A. Ferolito, Richard L. Bartolotti, Robert K. Yu, Ronald J. Melanson, Shailesh I. Shan, Sophie Nguyen, Sundari S. Mitra, Vinita Reddy, Vidyasagar Ganesan, and Willem J. de Lange. A 64-b Microprocessor with Multimedia Support. *IEEE Journal of Solid-State Circuits*, 30(11), November 1995.

[Lee95] Tak Kwan Lee. *A General Approach to Performance Analysis and Optimization of Asynchronous Circuits.* PhD thesis, California Institute of Technology, 1995. Technical report CS-TR-95-07.

[Lip94] Jonathan Brian Lipsher. The Asynchronous Discrete Cosine Transform Processor Core. Master's thesis, Electrical and Computer Engineering, University of California, Davis, 1994.

[LM92] Shih-Lien Lu and Lalit Merani. Micro Data Flow. In 4^{th} *Annual IEEE International ASIC Conference and Exhibit*, 1992.

[LSU89] Roger Lipsett, Carl F. Schaefer, and Cary Ussery. *VHDL, hardware description and design.* Kluwer Academic Publishers, 1989.

[MA93] Shannon V. Morton and Sam S. Appleton. Temporal transition graphs for event controlled systems. Report HPCA-ECS-93/12, Dept. of Electrical and Electronic Engineering, The University of Adelaide, Adelaide, South Australia, October 1993.

[MA94] Shannon V. Morton and Sam S. Appleton. ES2 Gates Library. Technical report, Department of Electrical and Electronic Engineering, University of Adelaide, SA 5006 Australia, June 1994.

[MAJL94] Shannon V. Morton, Sam S. Appleton, Andrew B. Johnson, and Michael J. Liebelt. Instruction Set Architecture of ECSTAC-P. Report HPCA-ECS-94/1, Dept. of Electrical and Electronic Engineering, The University of Adelaide, Adelaide, South Australia, 1994.

[MAL94] Shannon V. Morton, Sam S. Appleton, and Michael J. Liebelt. An Event Controlled Reconfigurable Multi-chip FFT. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 144–153, November 1994.

[MAL95] Shannon V. Morton, Sam S. Appleton, and Michael J. Liebelt. ECSTAC: A Fast Asynchronous Microprocessor. In *Asynchronous Design Methodologies*, pages 180–189. IEEE Computer Society Press, May 1995.

[Man95] R.T. Maniwa. An Alphabet Soup of Memory. *Integrated System Design*, May 1995.

[Mar81] Leonard R. Marino. General Theory of Metastable Operation. *IEEE Transactions on Computers*, C-30(2):107–115, February 1981.

[Mar90a] Alain J. Martin. Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[Mar90b] Alain J. Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. In William J. Dally, editor, *Sixth MIT Conference on Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[Mar92] Alain J. Martin. Tomorrow's Digital Hardware will be Asynchronous and Verified. In J. van Leeuwen, editor, *Information Processing 92, Vol. I: Algorithms, Software, Architecture*, volume A-12 of *IFIP Transactions*, pages 684–695. Elsevier Science Publishers, 1992.

[Max95] Clive Maxfield. Delay Effects Rule in Deep Submicron ICs. *Electronic Design*, June 1995.

[MBL+89a] Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The Design of an Asynchronous Microprocessor. In Charles L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 351–373. MIT Press, 1989.

[MBL+89b] Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The first asynchronous microprocessor: the test results. *Computer Architecture News*, 17(4):95–110, June 1989.

[McA92a] Anthony J. McAuley. Dynamic Asynchronous Logic for High-Speed CMOS Systems. *IEEE Journal of Solid-State Circuits*, 27(3):382–388, March 1992.

[McA92b] Anthony J. McAuley. Four State Asynchronous Architectures. *IEEE Transactions on Computers*, 41(2):129–142, February 1992.

[McL96] Jonah McLeod. Fab and Silicon Performance. *Integrated System Design*, August 1996.

[Men88] Teresa H.-Y. Meng. *Asynchronous Design for Digital Signal Processing Architectures.* PhD thesis, UC Berkely, 1988.

[Met96] Meta-Software Inc. *HSPICE Users Manual*, v.96-1 edition, 1996.

[Mic] Sun Microelectronics. microSPARC^{tm}-II Datasheet. Web Reference *www.sun.com/microelectronics/microSPARC-II.*

[Mil65] Robin Milner. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science. Springer-Verlag, 1965.

[Mis73] David Misunas. Petri Nets and Speed Independent Design. *Communications of the ACM*, 16(8):474–481, August 1973.

[MJCL97] Charles E. Molnar, Ian W. Jones, Bill Coates, and Jon Lexau. A FIFO Ring Oscillator Performance Experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems.* IEEE Computer Society Press, April 1997.

[MLCS93] Gary C. Moyer, Wentai Liu, Ralph K. Calvin, and Toby Schaffer. A High Speed CMOS Clock Shaper Using Wave Pipelining. Technical Report NCSU-VLSI-93/11, Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC, 1993.

[Mor97] Shannon V. Morton. *Fast Asynchronous VLSI Circuit Design Techniques and their Application to Microprocessor Design.* PhD thesis, University of Adelaide, 1997.

[Moy96] Gary C. Moyer. *The Vernier Techniques for Precise Delay Generation and Other Applications.* PhD thesis, Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC, 1996.

[MRW96] S. Moore, P. Robinson, and S. Wilcox. Rotary pipeline processors. *IEE Proceedings, Part E, Computers and Digital Techniques*, 143(5):259–265, September 1996.

[Mur89] Tadao Murata. Petri Nets : Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4), April 1989.

[Mye95] Chris J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits.* PhD thesis, Dept. of Elec. Eng., Stanford University, October 1995.

[nC96] M. A. Peña and J. Cortadella. 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.* IEEE Computer Society Press, March 1996.

[NK94] Christian D. Nielsen and Michael Kishinevski. Performance Analysis Based on Timing Simulation. In *Proc. ACM/IEEE Design Automation Conference*, pages 70–76, June 1994.

[Now93] Steven M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers.* PhD thesis, Stanford University, Department of Computer Science, 1993.

[Now96] S. M. Nowick. Design of a low-latency asynchronous adder using speculative completion. *IEE Proceedings, Part E, Computers and Digital Techniques*, 143(5):301–307, September 1996.

[NUK+94] Takashi Nanya, Yoichiro Ueno, Hiroto Kagotani, Masashi Kuwako, and Akihiro Takamura. TITAC: Design of a Quasi-Delay-Insensitive Microprocessor. *IEEE Design & Test of Computers*, 11(2):50–63, 1994.

[NYB97] Steven M. Nowick, Kenneth Y. Yun, and Peter A. Beerel. Speculative completion for the design of high-performance asynchronous dynamic adders. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems.* IEEE Computer Society Press, April 1997.

[Pav94] N. C. Paver. *The Design and Implementation of an Asynchronous Microprocessor.* PhD thesis, Department of Computer Science, University of Manchester, June 1994.

[Pee96] Ad M. G. Peeters. *Single-Rail Handshake Circuits.* PhD thesis, Eindhoven University of Technology, June 1996.

[Pet81] J.L. Peterson. *Petri Net Theory and Modelling of Systems.* Prentice-Hall, 1981.

[PH96] David A. Patterson and John L. Hennessy. *Computer Architecture : A Quantitative Approach.* Morgan-Kaufmann Publishers, Second edition, 1996.

[PP94] T. Pedron and J.F. Paillotin. *ES2 Design Rules ECPD07 – Layout and Electrical Rules, Dual Layer metal 0.7μm*, November 1994. Revision B.

[Pri91] Betty Prince. *Semiconductor Memories.* John Wiley, second edition, 1991.

[Puc90] Douglas A. Pucknell. *Fundamentals of Digital Logic Design : with VLSI circuit applicatons.* Prentice-Hall, 1990.

[Puc93] Douglas A. Pucknell. Event-Driven Logic(EDL) approach to digital system representation and related design processes. *IEE Proceedings, Part E, Computers and Digital Techniques*, 140(2), March 1993.

[Ric96] W. F. Richardson. *Architectural considerations in a self-timed processor design.* PhD thesis, Univerity of Utah, 1996.

[Røi94a] Per Torstein Røine. Asynchronous FIFO Buffer for Multicomputer Applications. Master's thesis, Department of Informatics, University of Oslo, 1994.

[Røi94b] Per Torstein Røine. Building Fast Bundled Data Circuits with a Specialized Standard Cell Library. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 134–143, November 1994.

[Røi96] Per Torstein Røine. A System for Asynchronous High-speed Chip to Chip Communication. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems.* IEEE Computer Society Press, March 1996.

[S+95] Aaron Sawdey et al. A GCC Machine Description for DLX, 1995. Web Address *www-mount.ee.umn.edu/mcerg/gcc-dlx.html.*

[Sak88] T. Sakurai. Optimization of CMOS Arbiter and Synchronizer Circuits with Submicron MOSFETs. *IEEE Journal of Solid-State Circuits*, 23(4):901–906, August 1988.

[Sch87] Dieter K. Schroder. *Advanced MOS Devices.* Modular Series of Solid State Devices. Addison-Wesley, 1987.

[Sch95] Steven E. Schulz. Timing Analysis Tools and Trends. *Integrated System Design*, November 1995.

[Sei80] Charles L. Seitz. System Timing. In Carver A. Mead and Lynn A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[Sei94] Jakov N. Seizovic. Pipeline Synchronization. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 87–96, November 1994.

[SK96a] Phillip M. Sailer and David R. Kaeli. *The DLX Architecture Handbook.* Morgan Kaufmann, 1996.

[SK96b] T. Schumann and H. Klar. An Asynchronous Up/Down Counter for Control of a Self-timed, Wave-pipelined Array Multipler. In *ACiD-WG Workshop*, September 1996.

[Sne85] Jan L. A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science.* Springer-Verlag, 1985.

[SSL+92] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, and A. Sangiovanni-Vincentelli. SIS : A System for Sequential Logic Synthesis. Technical Report UCB/ERL M92/41, Electronics Research Laboratory, Dept. of Electrical Engineering and Computer Science, May 1992.

[SSM94] 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.

[Sta94] Jørgen Staunstrup. *A Formal Approach to Hardware Design.* Kluwer Academic Publishers, 1994.

[Sut89] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[SY92] Christer Svensson and Jiren Yuan. High-Speed CMOS circuit technique. *IEEE Journal of Solid-State Circuits*, 27(3):382–388, April 1992.

[Tie95] José A. Tierno. *An Energy-Complexity Model for VLSI Computations.* PhD thesis, California Institute of Technology, 1995. Caltech technical report CS-TR-95-02.

[TMBL93] José A. Tierno, Alain J. Martin, Drazen Borkovic, and Tak Kwan Lee. An Asynchronous Microprocessor in Gallium Arsenide. Technical Report CS-TR-93-38, California Institute of Technology, 1993.

[TMBL94] José A. Tierno, Alain J. Martin, Drazen Borkovic, and Tak Kwan Lee. A 100-MIPS GaAs Asynchronous Microprocessor. *IEEE Design & Test of Computers*, 11(2):43–49, 1994.

[TV95] H.C. Torng and Stamatis Vassiliadis. *Instruction-Level Parallel Processors.* IEEE Computer Society Press, 1995.

[Ung69] S. H. Unger. *Asynchronous Sequential Switching Circuits.* Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[Ung97] Stephen H. Unger. *The Essence of Logic Circuits.* IEEE Press, second edition, 1997.

[VCGM90] P. Vanbekbergen, F. Catthoor, G. Goossens, and H. De Man. Optimized Synthesis of Asynchronous Control Circuits form Graph-Theoretic Specifications. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 184–187. IEEE Computer Society Press, 1990.

[VGCM92] Peter Vanbekbergen, Gert Goossens, Francky Catthoor, and Hugo J. De Man. Optimized Synthesis of Asynchronous Control Circuits from Graph-Theoretic Specifications. *IEEE Transactions on Computer-Aided Design*, 11(11):1426–1438, November 1992.

[VLGdM92] P. Vanbekbergen, B. Lin, G. Goossens, and H. de Man. A Generalized State Assignment Theory for Transformations on Signal Transition Graphs. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 112–117. IEEE Computer Society Press, November 1992.

[VYSS96] N. Vasseghi, K. Yeager, E. Sarto, and M. Seddighnezhad. 200-MHz Superscalar RISC Microprocessor. *IEEE Journal of Solid-State Circuits*, 31(11), November 1996.

[Wal90] David W. Wall. Limits of Instruction-Level Parallelism. Technical Note TN-15, Digital Western Research Laboratory, Palo Alto, CA, December 1990. Web Reference *www.research.digital.com/wrl/home.html.*

[WDF+97] J. V. Woods, P. Day, S. B. Furber, J. D. Garside, N. C. Paver, and S. Temple. AMULET1: An Asynchronous ARM Processor. *IEEE Transactions on Computers*, 46(4):385–398, April 1997.

[WE95] Neil Weste and Kamran Eshragian. *Principles of CMOS VLSI Design.* Addison Wesley, Second edition, 1995.

[Wei97] Reinhold Weicker. On the Use of SPEC Benchmarks in Computer Architecture Research. *Computer Architecture News*, 25(1):19–22, March 1997.

[WH91] Ted E. Williams and Mark A. Horowitz. A Zero-Overhead Self-Timed 160ns 54b CMOS Divider. *IEEE Journal of Solid-State Circuits*, 26(11):1651–1661, November 1991.

[Wil90] Ted E. Williams. Latency and Throughput Tradeoffs in Self-Timed Asynchronous Pipelines and Rings. Technical Report CSL-TR-90-431, Stanford University, August 1990.

[Wil91] Ted E. Williams. *Self-Timed Rings and their Application to Division.* PhD thesis, Stanford University, June 1991.

[XB97] Aiguo Xie and Peter A. Beerel. Symbolic Techniques for Performance Analysis of Timed Systems based on Average Time Separation of Events. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems.* IEEE Computer Society Press, April 1997.

[YBA96] K. Y. Yun, P. A. Beerel, and J. Arceo. High-Performance Asynchronous Pipeline Circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems.* IEEE Computer Society Press, March 1996.

[YD92] Kenneth Y. Yun and David L. Dill. Automatic Synthesis of 3D Asynchronous State Machines. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 576–580. IEEE Computer Society Press, November 1992.

[YDA+97] Kenneth Y. Yun, Ayoob E. Dooply, Julio Arceo, Peter A. Beerel, and Vida Vakilotojar. 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.* IEEE Computer Society Press, April 1997.

[Yea96] Kenneth C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, April 1996.

[YHJN95] 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.

[Yun94] Kenneth Yi Yun. *Synthesis of Asynchronous Controllers for Heterogeneous Systems.* PhD thesis, Stanford University, August 1994.