The Design and Implementation of an Asynchronous Microprocessor

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE

By

Nigel Charles Paver

Department of Computer Science

1994

Table of Contents

Chapter 1 : Introduction 17

1.1 Motivation 17

1.1.1 Global synchronization 17

1.1.2 Performance 18

1.1.3 Power consumption 18

1.2 Basic concepts 19

1.2.1 Timing model 19

1.2.2 Mode 19

1.2.3 Asynchronous signalling conventions 20

1.3 Objectives and thesis structure 22

1.3.1 Structure of the thesis 22

1.3.2 Author’s contribution 22

Chapter 2 : Related work 25

2.1 Automatic synthesis tools 25

2.1.1 CSP based compilation 25

2.1.2 Signal transition graphs 26

2.1.3 State machines 27

2.2 Other related work 27

2.3 Summary 28

2.3.1 Micropipelines 28

2.3.2 AMULET group Micropipelines 28

Chapter 3 : Micropipelines 29

3.1 Basic concepts 29

3.1.1 Event control modules 30

3.1.2 Metastability 31

3.1.3 Event-controlled storage element 32

3.2 Micropipelines 33

3.2.1 A Micropipeline FIFO 33

3.2.2 Micropipelines with processing 34

3.3 Event control module structures 34

3.3.1 Exclusive OR gate 35

3.3.2 Muller C-Gate 36

3.3.3 Transparent latch 37

3.3.4 SELECT block 37
3.3.5 Decision-Wait element 38
3.3.6 CALL block 39
3.3.7 TOGGLE 39
3.3.8 ARBITER 41
3.3.9 Capture-Pass latch 43
3.3.10 Cell layout 44
3.3.11 Implementation costs 44
3.4 Micropipeline implementation 46
3.4.1 The T-Latch Micropipeline 47
3.4.2 The Capture-Pass Micropipeline 49
3.4.3 Capture-Pass versus transparent latch area considerations ... 49
3.4.4 Micropipeline stage performance 51
3.4.5 Power considerations 53
3.4.6 Choosing an implementation 54
Chapter 4 : The asynchronous ARM 57
4.0.1 The ARM processor 57
4.0.2 Implementation challenges 57
4.0.3 Differences from the ARM6 58
4.0.4 Processor interface 58
4.1 Processor organization 60
4.1.1 Address interface 60
4.1.2 The register bank 60
4.1.3 The execution unit 60
4.1.4 The data interface 62
4.2 Pipeline organization and control 62
4.2.1 Dynamic pipeline structure 62
4.3 Instruction mapping 64
4.3.1 Data operation 65
4.3.2 Branch operation 66
4.3.3 Multiply operation 67
4.3.4 Load data operation 67
4.3.5 Store data operation 70
4.3.6 Block transfer operation 70
4.3.7 Exception entry operation 74
4.4 Instruction flow control 76
4.4.1 The ARM PC model and the PC pipeline 76

4.4.2 Condition code evaluation 76
4.4.3 Branch operations 77
4.5 Exception handling 79
4.5.1 Hardware interrupts 79
4.5.2 Data abort overview 81
4.5.3 Data abort signalling 82

Chapter 5 : The register bank 85
5.1 Register bank operation 85
5.1.1 Internal register structure 86
5.1.2 Register bank hazards 86
5.2 Write address storage 87
5.2.1 An asynchronous register lock FIFO 88
5.2.2 FIFO examination 89
5.2.3 Stalling reads 90
5.3 Asynchronous register bank design 90
5.3.1 Read and lock operations 90
5.3.2 Write operations 92
5.4 Additional features 93
5.4.1 Dual lock FIFO 93
5.5 Implementation 94
5.6 Future enhancements - register bypassing 96
5.6.1 Register through-passing 96
5.6.2 Last result re-use 97

Chapter 6 : Memory interface 99
6.1 Address interface 99
6.1.1 Load/Store multiple operation 101
6.1.2 Changing the PC value 101
6.1.3 PC loop deadlock 102
6.1.4 PC pipeline 103
6.1.5 Instruction overflow deadlock 103
6.1.6 PCpipe implementation 105
6.1.7 Incrementer 107
6.2 Data interface 108
6.2.1 Data in 108

Chapter 7 : Execution pipeline 111
7.1 The multiplier 111

7.2 The shifter 112
7.3 The ALU 112
7.3.1 Performance considerations 112
7.3.2 Implementation 113
7.4 The CPSR 113

Chapter 8 : Implementation 117

8.1 Design flow 117
8.1.1 Verification of the design 117
8.2 Complete organization 119
8.2.1 Datapath VLSI organization 121
8.3 Silicon layout 121
8.4 Test devices 123

Chapter 9 : Evaluation and further work 125

9.1 Design characteristics 125
9.1.1 Area overhead 126
9.1.2 Pipeline organization 127
9.2 Further work 128
9.2.1 Base technology 128
9.2.2 Processor organization 129
9.2.3 Tools 129
9.2.4 Test 130

Chapter 10 : Conclusions 131

10.1 Micropipelines 131
10.2 AMULET1 132
10.3 An asynchronous future? 132

Chapter 11 : Bibliography 133

Appendix A: Timing characteristics 143

A.1 Measurement conditions 143
SPICE timings for: CALL2 144
SPICE timings for: DMULLC2 146
SPICE timings for: DXor/DXNor 147
SPICE timings for: DWAIT2 148
SPICE timings for: MULLC2R 149
SPICE timings for: MULLC2 150

SPICE timings for: MULLC3R 151
SPICE timings for: TOGGLE 152
SPICE timings for: SELECT2 155
SPICE timings for: TLTCHR 156

Appendix B: The ARM processor 157

B.1 The ARM2 157
B.1.1 Instruction set 158
B.1.2 Organization 159
B.2 The ARM6 159

Abstract

A fully asynchronous implementation of the ARM microprocessor has been developed in order to demonstrate the feasibility of building complex systems using asynchronous design techniques. The design is based upon Sutherland’s Micropipelines and allows considerable internal asynchronous concurrency. The design exhibits several novel features including: a register bank design which maintains coherent register operation while allowing concurrent read and write access with arbitrary timing and dependencies, the incorporation of an ALU whose speed of operation depends upon the data presented, and an instruction prefetch unit which has a non-deterministic (but bounded) prefetch depth beyond a branch. The design also includes many complex features commonly found in modern RISC processors, such as support for exact exceptions, backwards instruction set compatibility and pipelined operation.

This thesis introduces the Micropipeline approach and discusses the design, organization, implementation and performance of the asynchronous ARM microprocessor which was constructed in the course of the work.

Declaration

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

Preface

The author’s first degree was in Electronics at the University of Manchester Institute of Science and Technology (UMIST). An MSc in System Design at the Computer Science Department of the University of Manchester was completed the following year. The author was then employed as a Research Assistant on the ESPRIT EDS project investigating the implementation of functional language execution mechanisms on a distributed store parallel machine. In January 1990 the author became a Research Associate working on the ESPRIT OMI-MAP project investigating the potential of asynchronous logic for low power applications. The author is currently employed as a Research Fellow working on the ESPRIT OMI-DE project which is continuing on from the work of OMI-MAP. This thesis reports the results of the work undertaken during the OMI-MAP project.

Acknowledgements

The design and implementation of an asynchronous microprocessor is a complex task and was a team effort so I appreciate the support given by all those involved. First I would like to thank Paul Day, the other full-time research associate working on this project, for the endless discussions and debates we had about the design and for his substantial contribution to the work. I would also like to thank Steve Furber for his inspired leadership of this project and his contribution to the design work. I acknowledge the contributions of Jim Garside, Viv Woods and Steve Temple to this project and appreciate the encouragement and support of other members of the AMULET group.

I am grateful for material support in various forms from Advanced RISC Machines Limited Acorn Computers Limited, Compass Design Automation Limited, and VLSI Technology Limited; in particular I would like to thank Harry Oldham and Dave Howard of ARM Ltd. for their advice concerning silicon layout issues. Thanks to Raj Chauhan, Neil Hastie, and everybody at GEC Plessey Semiconductors, involved in the translation and fabrication of the test devices.

The work described in this thesis was carried out as part of ESPRIT project 5386, OMI-MAP (the Open Microprocessor systems Initiative - Microprocessor Architecture Project), and I am grateful for this support from the CEC and the support of the OMI-MAP project officer Rosalie Zobel. The encouragement and support of the OMI-MAP consortium, and particularly the prime contractor, INMOS Limited, is also acknowledged.

List of Figures

Figure 1-1: Two-phase communication protocol 20
Figure 1-2: Four-phase communication protocol 21
Figure 3-1: Bundled data interface 29
Figure 3-2: The two-phase bundled data convention 30
Figure 3-3: Event logic library 30
Figure 3-4: Event-controlled storage element 32
Figure 3-5: Simple Micropipeline FIFO 33
Figure 3-6: Processing Micropipeline 34
Figure 3-7: Exclusive OR gates 35
Figure 3-8: Pseudo-static Muller C-Gate 36
Figure 3-9: Transparent latch 37
Figure 3-10: SELECT block 38
Figure 3-11: Decision-Wait element 39
Figure 3-12: Call block 39
Figure 3-13: TOGGLE high-level operation 40
Figure 3-14: TOGGLE element 40
Figure 3-15: NAND gate TOGGLE 41
Figure 3-16: ARBITER implementation 42
Figure 3-17: Robust two-phase arbiter 43
Figure 3-18: Capture-Pass implementation 43
Figure 3-19: Examples of cell layout 45
Figure 3-20: Cell silicon area comparison 46
Figure 3-21: Simple transparent latch Micropipeline stage 47
Figure 3-22: Simple transmission gate latch 47
Figure 3-23: Transparent latch Micropipeline control 48
Figure 3-24: Capture-Pass Micropipeline control 49
Figure 3-25: Relative size of Capture-Pass v T-Latch Micropipeline 50
Figure 3-26: Fast-forward Micropipeline control 53
Figure 3-27: Micropipeline fork 54
Figure 3-28: Blocking Micropipeline control 55
Figure 4-1: The processor interface 59
Figure 4-2: Processor organization 61
Figure 4-3: Pipeline organization and control 63
Figure 4-4: Dynamic pipeline structure 64
Figure 4-5: Data operation datapath activity 65
Figure 4-6: Branch & branch and link datapath activity 66
Figure 4-7: Multiply datapath activity 68
Figure 4-8: Load register datapath activity 69
Figure 4-9: Store register datapath activity 71
Figure 4-10: Load/Store multiple datapath activity 73
Figure 4-11: Exception entry datapath activity 75
Figure 4-12: Instruction/PC matching 77
Figure 4-13: Interrupt and PC colour entry into the instruction stream 80
Figure 4-14: Hardware interrupt timing 81

Figure 4-15: Data abort entry into the primary decode 83
Figure 5-1: Register bank operation 85
Figure 5-2: Register bank internal structure 86
Figure 5-3: Decoder arrangement 87
Figure 5-4: Write address storage 88
Figure 5-5: Lock detection 88
Figure 5-6: Lock interrogation 89
Figure 5-7: Read lock gating 90
Figure 5-8: Register bank organization 91
Figure 5-9: A Petri Net model of the read-lock sequencing 92
Figure 5-10: Dual lock FIFO configuration 94
Figure 5-11: Register bank silicon implementation 95
Figure 5-12: Improved register write circuitry 97
Figure 6-1: The address interface 100
Figure 6-2: PC loop potential deadlock 102
Figure 6-3: Staggering of PC pipe to achieve correct R15 value 103
Figure 6-4: PC pipe behaviour across a branch 104
Figure 6-5: Potential processor deadlock 105
Figure 6-6: The PC pipelines 106
Figure 6-7: Data abort PC storage 106
Figure 6-8: Simple ripple-carry incrementer cell 107
Figure 6-9: Data-in organization 108
Figure 7-1: Execute pipeline 111
Figure 7-2: ALU carry propagation statistics 114
Figure 7-3: ARM6 v AMULET1 ALU area 115
Figure 7-4: CPSR structure 116
Figure 8-1: Design flow 118
Figure 8-2: Internal organization 120
Figure 8-3: Organization of the datapath and control dependencies 122
Figure 8-4: AMULET1 1.2 micron physical layout 123
Figure 8-5: AMULET1: 0.7 multi-project die organization 124
Figure 8-6: AMULET1: 0.7 micron multi-project die 124
Figure 9-1: Svensson style latch 129
Figure B-1: The ARM2 register organization 157
Figure B-2: Program counter and program status word 158
Figure B-3: ARM2 block diagram 160
Figure B-4: The ARM6 register organization 161

“What I did not realise is how much simpler the control circuits are in a parallel asynchronous machine than in a serial synchronous one.”

M.V.Wilkes.

Memoirs of a Computer Pioneer, p164
MIT Press ISBN 0-262-23122-0
1985

Chapter 1 : Introduction

Most current digital design is based upon a synchronous approach. However, recently there has been renewed interest in asynchronous design styles where instead of a global clock to regulate operation, the subsystems of the design communicate with each other at arbitrary times when they wish to exchange information. Much of the recent work in asynchronous logic design has been motivated by perceived difficulties in certain aspects of synchronous VLSI[1] design. At present these difficulties are being overcome but the cost is increasing as silicon geometry sizes decrease and clock frequencies and the degree of on-chip integration increase.

1.1 Motivation

Asynchronous design styles may offer reduced cost solutions to several of the emerging synchronous design difficulties. The three main areas that may benefit most from an asynchronous design approach are global synchronisation, performance and power consumption. To understand why asynchronous designs may offer advantages in these areas it is necessary to understand the nature of the problems and how they are currently being resolved (and the cost associated with doing so in a synchronous environment).

1.1.1 Global synchronization

With the decrease in process feature size and the increase in the degree of on-chip integration it is becoming increasingly difficult to maintain the global synchronization required in a clocked system. The difficulty lies in distributing the clock signal across the silicon in such a way that all elements receive a transition of the clock at the same time. The maximum time difference between any two parts of the circuit observing the same clock transition is known as the clock skew. The next clock transition cannot be allowed to occur until the previous transition has propagated to all parts of the circuit. If the clock skew is large then the clock period must be extended to ensure correct operation and as a result the maximum frequency is limited by the on-chip skew.

State of the art designs [DEC92] have demonstrated that it is possible to engineer circuits to overcome these problems, but the cost is high. For example, in the Alpha processor (21064-AA [DEC92a]) about a third of the silicon area is devoted to the clock drivers. The designers of the circuit carefully modelled the delay through the clock distribution network [Dobb92] to ensure that the clock skew was small enough to allow 200MHz operation.

Asynchronous circuits have no global clock so there is no global synchronization constraint to satisfy and the complex detail design of driver networks is not required.

1. Very Large Scale Integration

1.1.2 Performance

A characteristic of normal synchronous design it that it is optimized for worst-case conditions. The minimum clock period (and hence maximum frequency) is constrained by the operation that takes the longest time to complete. The clock frequency is fixed so that every cycle is long enough to allow for the worst-case operation even though, typically, the average case could be handled in a much shorter time. The time variation between worst-case and typical operations is usually significant, so optimizing a circuit for typical rather than worst-case operations has advantages which are not available to the synchronous designer.

The speed of a particular operation is affected by a number of independent factors:

- Variations in the silicon processing of CMOS circuits leads to variations in transistor strengths between limits. The worst-case is when both n- and p-transistors are slow and the process is classified as *slow-slow*. Transistors from a *typical* process usually operate at approximately twice the speed of transistors from a *slow-slow* process and transistors from a *fast-fast* process usually show a factor four increase in speed over *slow-slow* transistors. As a process matures a higher percentage of the devices fabricated fall into the *typical* category and the process variations are much reduced. However, many high performance processors take advantage of leading edge technologies where the process variation may be high.

- Logic functions may have certain input data values that require more time to evaluate than the average case. For example, a ripple-carry adder where the carry has to ripple through all the bit positions requires more time for the result to become stable than a carry that only ripples across a small part of the data word.

- The power supply voltage and temperature of a CMOS circuit affects its speed. The operation becomes slower with a decrease in supply voltage or an increase in temperature and becomes faster if the temperature is lowered or the supply voltage is raised (there are limits to the extent of voltage and temperature variations to ensure the circuits can still operate).

For a synchronous system, the fixed clock period must be set to accommodate the situation where the worst-case of all these factors exists at the same time.

With an asynchronous system it is possible to construct circuits optimized for the typical case; worst-case operations simply take longer when required (there is no fixed clock period during which the operation must be completed).

1.1.3 Power consumption

Power consumption is becoming increasingly important in the emerging market of hand held portable computing equipment [Lind92], where battery life is at a premium. Power consumption is also becoming a problem in high performance RISC[1] processors with recent designs dissipating 20 - 30 Watts [DEC92, Sun92], which leads to challenges in packaging and system design to remove the generated heat.

1. Reduced Instruction Set Computer

In CMOS, the power dissipated is proportional to the frequency of the clock [Eshr89], so as the clock frequency increases the power dissipated also increases accordingly. Decreasing the power supply voltage from 5V to 3V reduces the power by a factor of three, but there are limits to how low the supply voltage can go before the device stops functioning correctly.

The power reduction offered by the shrinking of process geometries is usually offset by an increase in clock frequency and an increase in the functionality integrated on a single device. If the rate of increase in power consumption remains unchecked then this will shortly lead to a power (and performance) limit restricted by heat dissipation. A recent design [Joup93, Hamb92]) has demonstrated a packaging technique involving a thermosiphon capable of cooling a 150W device. This again shows that there if there is sufficient demand an engineered solution can be obtained but again the cost is high. Asynchronous design styles may offer another avenue to reducing the power consumption.

In a synchronous system, transitions of the clock are distributed across the entire chip on every cycle, so all parts of the circuit are activated and dissipate power whether they are needed or not. For example, in a microprocessor, the floating point unit may not be required for a particular instruction but it must still be clocked on every cycle. Techniques have been proposed for disabling the clock in areas of the circuit that are not in use by adding logic functions into the clock buffers. This makes the problem of global synchronization even more difficult, so it is not a feasible solution for systems with high clock frequencies.

An asynchronous system, on the other hand, only activates a particular part of the circuit when it is actually required and so does not dissipate any power in subcircuits that are not required.

1.2 Basic concepts

There are a few key concepts fundamental to the understanding of asynchronous circuits: the timing models used, the mode of operation and the signalling conventions.

1.2.1 Timing model

Asynchronous circuits are classified according to their behaviour with respect to circuit delays. If a circuit functions correctly irrespective of the delays in the logic gates and the delays in the wiring it is known as *delay-insensitive*. A restricted form of this circuit known as *speed-independent* allows arbitrary delays in logic elements but assumes zero delays in the interconnect (i.e. all interconnect wires are equi-potential). Finally, if the circuit only functions when the delays are below some predefined limit the circuit is known as *bounded-delay.*

1.2.2 Mode

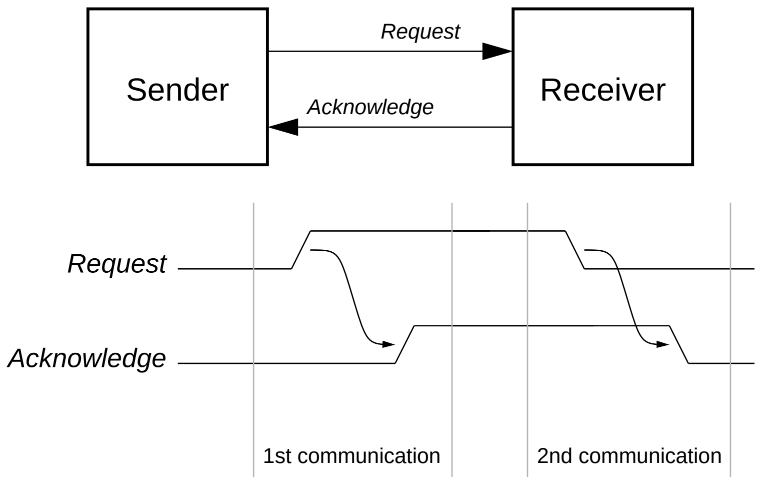
Asynchronous circuits can operate in one of two modes. The first is called *fundamental* mode and assumes no further input changes can be applied until all outputs have settled in response to a previous input. The second, *input/output* mode, allows changes to the inputs while the asynchronous circuit is still generating the outputs.

1.2.3 Asynchronous signalling conventions

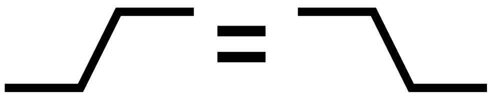
A communication between two elements in an asynchronous system can be considered as having two or four phases of operation and a single bit of information can be conveyed on either a single wire or a pair or wires (known as dual-rail encoding).

Two-phase

In a two-phase communication the information is transmitted by a single transition or change in voltage level on a wire. Figure 1-1(a) shows an example of two-phase communication.



(a) Communication protocol



(b) transition direction not important

Figure 1-1: Two-phase communication protocol

The sender initiates the communication by making a single transition on the request wire; the receiver responds by making a single transition on the acknowledge wire completing the two phases of the communication. The electrical level of the wires contains no information, only a transition is important and rising or falling transitions are equivalent (see figure 1-1(b))

There is no intermediate recovery stage, so that if the first communication resulted in a transition from Low to High the new communication starts with a transition High to Low (see figure 1-1(a), 2nd communication).

Four-phase

With four-phase communication two phases are active communication while the other two permit recovery to a predefined state. Figure 1-2 shows an example of four-phase communication; in this example all wires are initialised to a logical Low level.

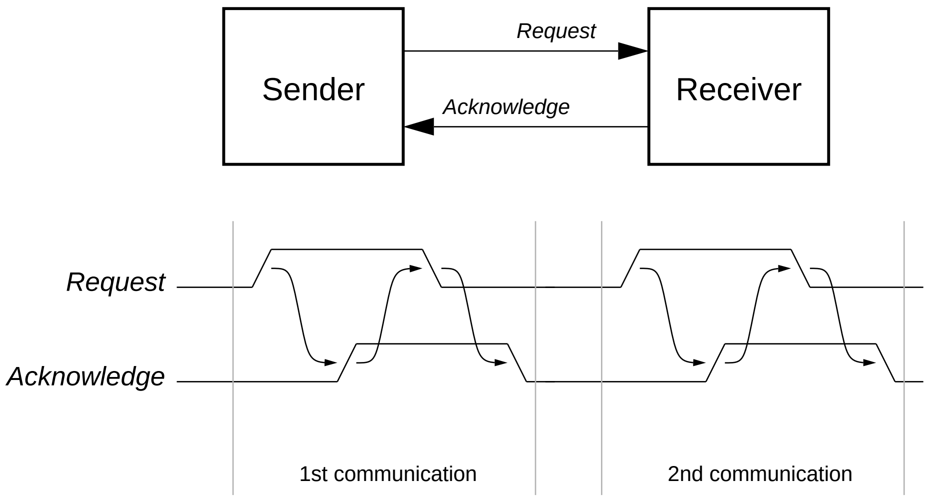


Figure 1-2: Four-phase communication protocol

The communication is initiated by the sender changing the request wire to a High level to indicate that it is active. The receiver responds by changing the acknowledge wire to a High level also. The sender observes this change, indicating that the communication has been successful, and then changes the request wire back to Low to indicate it is no longer active. The receiver completes the fourth phase of the operation by changing the acknowledge wire back to a Low level to indicate that it too has become inactive.

After completing the four phases of a single communication, the voltage levels on the wires have returned to their initial value (c.f. two-phase, where this is not the case).

Single-rail encoding

A single-rail circuit encodes information in a conventional level encoded manner. One wire is required for each bit of information. If the information is a data value, then a typical encoding would use a High (*Vdd*) level to correspond to a logic ‘1’ and a Low level (*Vss*) to represent a logic ‘0’.

Dual-rail encoding

A dual-rail circuit requires two wires to encode every bit of information. Of the two wires, one represents a logic ‘0’ and the other represents a logic ‘1’. In any communication an event occurs on either the logic ‘0’ wire or the logic ‘1’ wire. There cannot be an event on both wires during any single communication (a value cannot be ‘0’ and ‘1’ at the same time in a digital system). Similarly, in every communication there is always an event on one of the two wires of each bit (a value has to be ‘0’ or ‘1’). It is therefore possible to determine when the entire data word is valid because an event has been detected on one of the dual rails of every bit in the data word. Thus timing information is implicit with the data to indicate its validity. The event that is transmitted on one of the dual rails can either be two-phase or four-phase.

There are various combinations of two-/four-phase and single-/dual-rail protocols that can be used. Four-phase, dual-rail is popular for delay-insensitive asynchronous design styles. The research described in this thesis employs a combination of styles. The control circuitry is predominately two-phase, single-rail, although four-phase is used where it is more efficient to do so. Dual-rail is also used but only in a few specialised applications. The datapath part of the design uses standard single-rail logic to implement the functional units.

Overall the design adheres to the bounded-delay timing model (although some parts may be considered delay-insensitive) and its pipeline stages operate in fundamental mode.

1.3 Objectives and thesis structure

The objective of this work is to investigate whether it is possible to build commercially realistic complex circuits using an asynchronous design style and then assess what advantages the circuits designed may offer. To demonstrate the asynchronous methodology adopted (Micropipelines), an asynchronous implementation of the ARM[1] processor was designed and fabricated on a CMOS process.

1.3.1 Structure of the thesis

A survey of recent work in the area of asynchronous logic design is presented in chapter 2 and an in-depth description of the Micropipeline design methodology and implementation is described in chapter 3. The organizational features of the asynchronous ARM processor are described in chapter 4, including how the ARM instruction set is mapped onto an asynchronous organization. Chapters 5, 6 and 7 provide more detail about the register bank, memory interface and the execution pipeline respectively. Chapter 8 brings together the material of chapters 5, 6 and 7 and shows how the complex organization is mapped onto silicon, both in terms of design flow and VLSI organization. Chapter 9 evaluates the design and proposes further work (much of which is already underway).

Chapter 10 describes the conclusions that have been drawn so far from this work. Chapter 11 is the bibliography. Appendix A contains the SPICE characterisation figures for the event control modules described in chapter 3 and appendix B gives a brief overview of the ARM architecture.

1.3.2 Author’s contribution

The design and implementation of an asynchronous processor is a complex task undertaken by a group of researchers (two full-time Research Associates, including the author, and three members of academic staff). This section seeks to clarify the contribution of the author to the work described in this thesis.

Chapter 3 contains mainly background information about Micropipelines and the implementations that were chosen. The author contributed to the implementation and

1. The ARM (Advanced RISC Machine) is a 32-bit RISC processor (see appendix b)

characterisation of the cell library (although most of the cells were based on existing designs) and also performed the comparison of Micropipeline latch styles (section 3.4).

Chapter 4 contains a high-level architectural overview of the complete processor. This work was undertaken by the whole group but with substantial input from the author.

Chapter 5 and chapter 6 (with the exception of the incrementer in section 6.1.7) contains work primarily undertaken by the author.

Chapter 7 contains a brief description of the execute pipeline. The author did not contribute directly to this work and it is included for completeness in order that all aspects of the asynchronous microprocessor design may be fully described.

Chapter 8 describes the design flow used and the silicon implementation. The author contributed to the design flow by writing a test vector translation tool and the bundle checker and had a substantial contribution to the silicon layout.

Chapter 2 : Related work

The Macromodules project [Clar67] in the early 1960’s demonstrated the modular design of asynchronous circuits and the ease with which designs could be put together. The early work of Miller [Mill65], Molnar [Clar74, Moln83, Moln85], Seitz[Seit70, Seit80], Unger [Unge59, Unge69], Huffman[Huff54] and Keller[Kell74] laid the foundations on which most of the recent research in asynchronous design is based. The asynchronous phenomenon of metastability was discovered during the early work [Chan73, Kinn76].

As the general aspects of asynchronous design methodologies are beyond the scope of this thesis a brief description of current asynchronous design styles is presented here; a more in-depth survey and bibliography are presented elsewhere [Gopa90, Hauk93, Asyn93].

2.1 Automatic synthesis tools

There are three predominate styles of automatic synthesis of asynchronous circuits. The first is based on compiling from a high level language to a VLSI circuit. The second technique uses a graphical description as the starting point and the third uses asynchronous finite state machines to describe the circuit.

2.1.1 CSP based compilation

A number of compilation schemes derive asynchronous VLSI circuits from high-level languages based upon Communicating Sequential Processes (CSP) [Hoar78, Hoar85] and Occam [Inmo83] derivatives.

Brunvand [Brun89, Brun91] presents a technique for compiling a subset of Occam into delay-insensitive control circuits with bounded-delay datapath elements. The target of the design flow [Brun91a] is a set of control and data modules rather than a transistor level circuit. Circuit transformations can be applied after compilation to optimize the resulting circuit in a similar manner to peephole optimizers in a software compiler. The VLSI circuits constructed using this technique have been mapped onto CMOS, Field Programmable Gate Arrays (FPGA) and Galium Arsenide technologies [Brun91b, Brun91c, Brun92]. A simple RISC processor has been implemented on multiple FPGA elements to demonstrate the practicality of the approach [Brun93].

Martin [Mart86, Mart89, Mart90] and Burns [Burn87, Burn88] describe a technique for translating from a “program notation” based on CSP and Dijkstra’s guarded-commands [Dijk76] to a four-phase delay-insensitive circuit. The synthesis method has been demonstrated with numerous circuit examples [Mart85, Mart85a, Mart85b]. An asynchronous “RISC style” microprocessor has been developed [Mart89a, Mart89b] that demonstrated the feasibility of the approach, however the processor constructed was a very simple 16-bit machine with no support for the difficult areas of hardware interrupts and exact exceptions.

Van Berkel at Philips Research also describes a compilation system [vBer88, vBer88a, Nies88] based upon CSP and Dijkstra's guarded-command language. The term "VLSI programming" is introduced to describe the process of writing a program to generate a VLSI circuit and the language *Tangram* [vBer91] is used as a VLSI programming language.

Compilation begins with the translation of the Tangram program into an intermediate form known as *handshake circuits* [vBer92]. A handshake circuits is a network of components connected together by point-to-point channels which interact only by transition signalling along the channels (there are no global variables). The handshake circuit is converted into a netlist of standard-cell VLSI modules for final silicon layout.

The resulting VLSI circuits use a delay-insensitive, four-phase, dual-rail protocol for communication between components (the intermediate form handshake circuits use a two-phase delay-insensitive protocol).

The Tangram compilation system is a well integrated design system which incorporates a suite of tools which include:

- A translator from Tangram to handshake circuits and behaviourally equivalent C programs.
- An analyser which produces circuit level statistics.
- A compiled C-Code simulator for coarse timing.
- A converter into VHDL and a VHDL simulator for detailed timing.
- A standard-cell net-list generator and a standard-cell layout package.
- A test-trace generator

The system has been used to generate a number of VLSI circuits [Saei88, Kess90, Kess90a, Kess91, Kess92].

To address the area overhead associated with the Tangram use of dual-rail encoding for datapath elements further work is being undertaken in the OMI-EXACT project, in conjunction with the AMULET group at the University of Manchester [Edwa93], to investigate the use of two-phase, bounded-delay techniques to reduce the area overhead.

Gopalakrishnan and Akella present a design environment for the specification, simulation and synthesis of asynchronous circuits [Gopa93, Akel91], their specification language is also based upon CSP.

2.1.2 Signal transition graphs

A methodology for synthesizing speed-independent circuits from State Transition Graphs (STGs) was proposed by Chu [Chu85, Chu86a, Chu86b, Chu87]. STGS are similar to Petri Nets whose transitions are labelled with signal names and whose places form the arcs of the graph. When a transition fires in an STG the associated signal in the circuit changes. By restricting the allowable structure of an STG it is possible to generate a state assignment graph from which a circuit may be realised. A technique known as *Contraction* was developed to help implement STG circuits without the exponential explosion in complexity often associated with Petri Net circuits.

Several other researchers use the STG format to described asynchronous circuits and they have developed algorithms for STG transformations and synthesis [Lin91, Lin91a, Lin92a, Meng89, Vanb90a, Vanb90b, Yako92].

2.1.3 State machines

Davis, Coates and Stevens describe a collection of synthesis tools (MEAT) for generating hazard free asynchronous finite state machines [Coat93]. The specification is a state diagram with a restriction on the allowable input changes (known as burst-mode). The tool generates a schematic at the complex gate transistor level (c.f. CSP compilation methodologies which target predefined modules). The tool has been used to develop a complex communication chip with over 300,000 transistors [Coat93a]. Dill's verifier [Dill89] has been used to analyse the resulting circuits for hazards.

Nowick and Dill have presented a technique for the automatic synthesis of asynchronous state machines using a local clock [Nowi91, Nowi91b, Nowi92]. Together with Yun they have also proposed a technique for the synthesis of 3D state machines [Yun92] and an extension to the specification of burst-mode to allow more concurrency [Yun92b]. Much of the work of Davis, Dill and Nowick is being integrated into a single tool called Stetson [Davi93].

2.2 Other related work

Myers and Meng present a synthesis method that uses timing constraints to generate a timed asynchronous circuit [Myer92, Myer93]. Circuit examples are given to demonstrate the advantages of this approach in comparison to speed-independent approaches. Beerel and Meng [Beer92] describe a CAD tool for the synthesis of speed-independent asynchronous control circuits that use only basic gates. Lavagno [Lava92] describes a design technique where control circuits are designed using synchronous techniques and extra logic is added to remove hazards.

Josephs and Udding describe an algebraic approach to the design of delay-insensitive circuits [Jose90, Jose91] which allows the functional behaviour of primitive delay-insensitive elements to be captured by algebraic expressions. Their so-called *D-I Algebra* allows the designer to specify the circuit and the constraints that must be met by the environment precisely. The algebra also supports verification of the design against its specification. Simple designs using the algebra have been demonstrated [Jose90, Jose92a].

Ebergen [Eber91] also presents a formal approach to the design of delay-insensitive circuits from a specification based upon CSP and Dijkstra's guarded-command language. A modulo-N counter circuit developed using this technique exhibited a bounded response time and bounded power consumption [Eber92].

Rem [Rem90] provides a precise mathematical definition of delay-insensitivity, decomposition and speed-independence and the issues of using delay-insensitive circuits are discussed by Martin [Mart90a] and van Berkel [vBer92a].

Williams addresses the issues of latency and throughput trade-offs in self-timed speed-independent pipelines [Will90] and also describes a 54-bit self timed CMOS division implementation [Will91, Will91a].

2.3 Summary

The recent work described in this chapter has focused on novel design and verification techniques with emphasis on mathematical approaches to the automatic synthesis of asynchronous circuits. These techniques suffer from several drawbacks:

- They are limited to the size, type and complexity of circuit they can process.
- The resulting circuits often carry a large area and performance overhead.

It is the authors opinion that these techniques had not matured sufficiently to synthesise an asynchronous implementation of an ARM processor where the results are of practical proportions. Therefore a less formal engineering approach, based upon *Micropipelines*, was used to build the asynchronous implementation of an ARM processor.

2.3.1 Micropipelines

Sutherland [Suth89] describes a methodology called Micropipelines for the design of asynchronous systems using a two-phase bounded-delay protocol (Micropipelines are described in detail in chapter 3).

Gopalakrishnan investigates some unusual Micropipeline circuits [Gopa93] and the dynamic reordering of instruction sequences using a modified Micropipeline [Gopa92]. The AMULET group at Manchester University have investigated various aspects of Micropipeline designs.

2.3.2 AMULET group Micropipelines

Furber, Paver and others give an overview of the design of the asynchronous ARM processor [Furb92, Furb93a, Furb93b, Furb93c, Furb94, Pave93]. Day, Garside and Paver also discuss detailed aspects of the design [Day92, Day93, Gars92, Gars93, Pave91, Pave92a, Pave92b]. Other researchers within the AMULET group at Manchester University are constructing a bipolar implementation of the asynchronous ARM processor [Kell93]. The use of Micropipeline design styles in cost sensitive consumer products is being investigated in conjunction with Philips Research under the OMI-EXACT project [Farn93, Farn93a, Farn94]. Work on the OMI-HORN project at Manchester is investigating the application of asynchronous techniques for low cost and low power microprocessors in conjunction with INMOS Ltd. MSc research within the AMULET group has investigated cache structures suitable for an asynchronous processor [Mehr92], the high-level modelling of Micropipelines [Tan92], and the architectural features desirable for low power asynchronous microprocessors [Ende93]. PhD research is investigating the modelling of the asynchronous ARM microprocessor using Occam as a description language [Theo93].

Chapter 3 : Micropipelines

In the Turing Award lecture of 1988 Ivan Sutherland outlined a framework for designing asynchronous circuits [Suth89]. The lecture, entitled “Micropipelines”, included a description of a library of circuits that can be used to build asynchronous control structures and a technique for encapsulating asynchronous sub-circuits using a predefined interface. These basic sub-circuits can then be connected together to form asynchronous pipeline systems whose communication protocol was also described in the lecture.

3.1 Basic concepts

Micropipelines use a two-phase bundled data interface as illustrated in figure 3-1 below. This interface has an arbitrary number of data bits accompanied by two signalling wires called Request (*Req*) and Acknowledge (*Ack*). The communication protocol used by the sender and receiver is illustrated in figure 3-2. Here the sender prepares the data during its active phase (denoted by the grey area) and, when the data is valid (denoted by the white area), signals this to the receiver by generating a transition (in the first case from Low to High) on the *Req* wire. The receiver then begins to process the data. When the receiver has finished with the data on its input the sender is signalled to indicate that the data has been received. This acknowledgement is transmitted by generating a transition on the *Ack* wire (the transition in this case is also Low to High). On receiving an *Ack* the sender can remove the data and begin preparing the next value. During the next cycle, after the sender has prepared the next set of data, the request wire this time makes a transition from High to Low. In keeping with the two-phase philosophy the direction of the transition is not important, only that one has occurred. Again, when the receiver has processed the data, it signals back to the sender with a transition (also from High to Low) on the acknowledge wire.

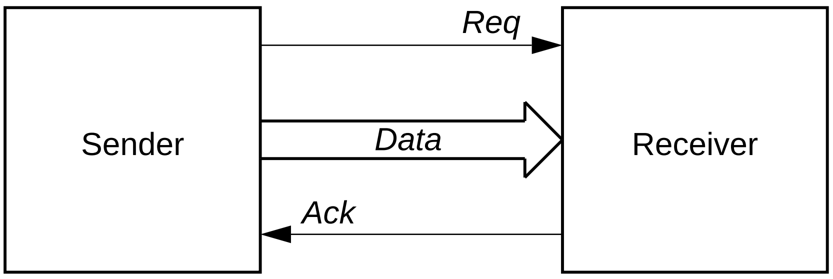


Figure 3-1: Bundled data interface

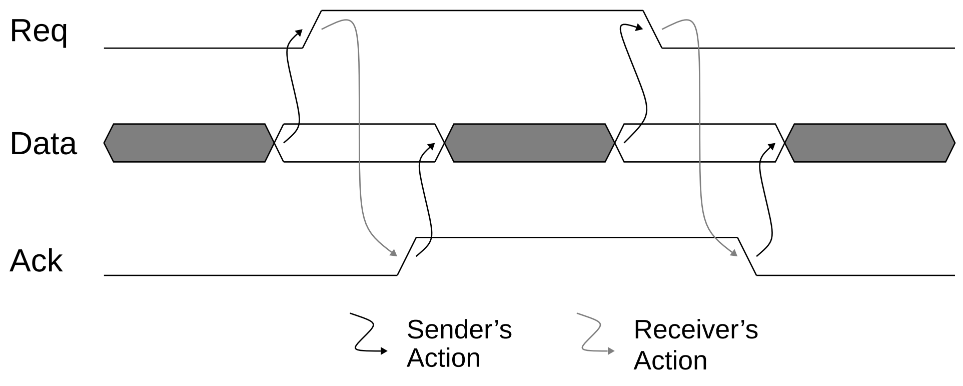


Figure 3-2: The two-phase bundled data convention

Micropipelines are considered bounded-delay because the data are constrained to be valid before the request and therefore the delay on the data must be less than the delay on the request i.e. it is bounded. The requirement for the data to be valid before a request is issued is known as the bundled-data delay constraint.

3.1.1 Event control modules

To ease the design of circuits using transition signalling, Sutherland proposed a library of basic building blocks, as shown in figure 3-3. The first element shown here is the exclusive OR gate (**XOR**). This circuit acts as the **OR** function for events in that an event arriving on either input will cause an event on the output. For correct operation the environment[1] must

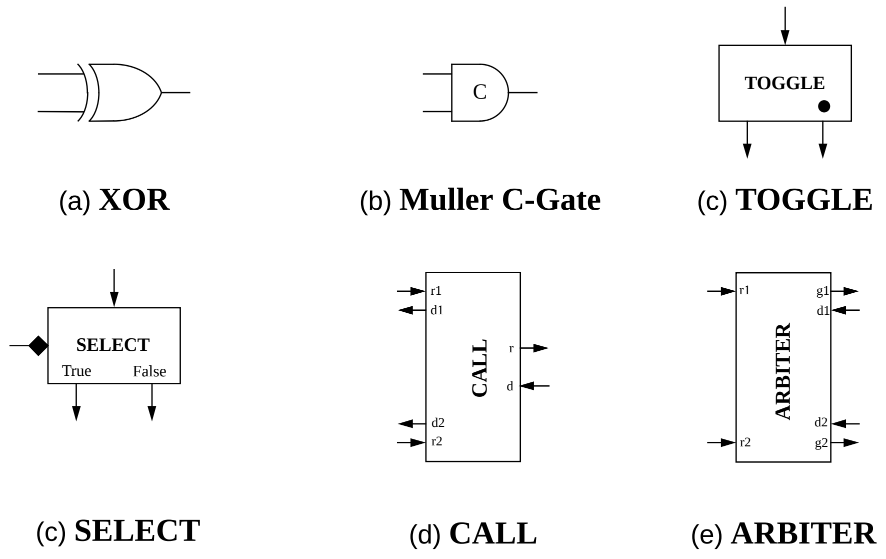


Figure 3-3: Event logic library

1. The environment is the term used to describe the circuits within which an event module is placed

ensure that events do not arrive simultaneously on both inputs. The **XOR** elements are often called MERGE elements because they are used to merge two event streams into one.

The Muller C-Gate [Mill65] (referred to as the **C-Gate** for brevity), figure 3-3(b), acts as an AND function for transition events. Here each input must receive an event before an event is generated on the output. The **C-Gate** is also known as a RENDEZVOUS element because it only enables events to continue when there has been an event on both inputs. Various forms of **C-Gate** are useful with different numbers of inputs some of which may be initialised to an active state (see section 3.3.2). The primed inputs are indicated by placing on small circle on them.

The **TOGGLE** (figure 3-3(c)) steers incoming events to alternate outputs. After initialisation the first input event is steered to the output marked with a dot (*Dot*). The next event is steered to the other output (*Blank*) and then the cycle repeats for further input events.

The **SELECT** block (figure 3-3(c)) also steers events to one of two outputs. The destination here is determined by the value of the Boolean select signal (indicated by the diamond in figure 3-3(c)). A High input on the Boolean select line causes the input event to be steered to the *True* output, a Low input causes the event to exit via the *False* output. The Boolean signal must be set up before the arrival of the event and must not change close to the time the input event arrives; this is a constraint which the environment must satisfy to ensure reliable operation.

If two circuits share access a single sub-circuit the interaction can be controlled by a **CALL** block (figure 3-3(d)), with the two circuits submitting requests on *r1* & *r2* respectively. The circuit which submits a request (*r1* & *r2* must be mutually exclusive) has the request routed to the request out (*r*) and on to the sub-circuit. When the sub-circuit has completed processing it returns an acknowledge (*a*) to the **CALL** block where it is steered back to the correct calling circuit, either *d1* or *d2.*

The **CALL** block is analogous to a procedure call in software where a common subroutine is called from two different places in the main program. The block is configured so that the acknowledge is steered back to the correct calling circuit; the software equivalent of this action is returning to the stored return address.

The **ARBITER** (figure 3-3 (e)) is used to control the interaction between two asynchronous event streams. As the two streams can present requests at arbitrary relative times, the arbitration logic is inherently prone to metastability. Internally the **ARBITER** must be able to handle metastable states while still presenting valid logic levels at its interface.

3.1.2 Metastability

Metastability is the phenomenon whereby non-digital logic values are seen at the output of a state storing element caused by the input to the element changing too close to the sample point of the input. The metastable value on the output can persist for an arbitrary time before eventually settling to one of the valid digital values.

The behaviour of a metastable system can be modelled mathematically. In particular the probability of the non-digital values persisting on the output can be shown to be a negative exponential function of time [Cour75, Hors89].

In a traditional clocked system, synchronisation is performed by sampling an input, waiting a time calculated to ensure that the probability of the metastability persisting is extremely small, and then sampling the output to determine its value. All synchronous systems have a finite (but small) probability of synchronisation failure because the output sample point is fixed by the clock period. There is a trade-off between the reliability of a system and the length of time allowed for metastability to be resolved.

In an asynchronous system it is possible to detect metastability and delay any output until the metastability is resolved. The **ARBITER** proposed by Sutherland takes two (possibly simultaneous) input requests, arbitrates between them, and when a definite decision has been made issues a grant signal to whichever output was chosen. The **ARBITER** can be combined with the **CALL** block to enable the two processes which are not mutually exclusive to share a common sub-process.

3.1.3 Event-controlled storage element

The blocks described above can be used to compose transition signalling control structures. To construct a complete system also requires some state storage elements. Sutherland proposed a latch based storage element with two transition signalling control wires termed “capture” and “pass”. Figure 3-4 shows one of Sutherland’s implementations and the symbol used to denote it.

During initialisation the latch is reset to a transparent state where the input is connected through to the output. When a capture event occurs the input is disconnected from the forward inverter forming the path to the output. The output of this inverter is now connected via an inverter back to its input, forming a state-retaining loop. This loop is still connected to the output, which therefore reflects the previous or “latched” value of the input and does not change with subsequent input changes. The input is now connected to the lower forward inverter. When a “pass” event arrives the output is switched from the upper inverter loop previously described to the lower inverter pair once again allowing the input to flow through to the output but this time through the lower inverter. The latch is again transparent and the **Capture-Pass** sequence may repeat.

The **Capture-Pass** latch and the transition signalling library building blocks can be used to form Micropipelines.

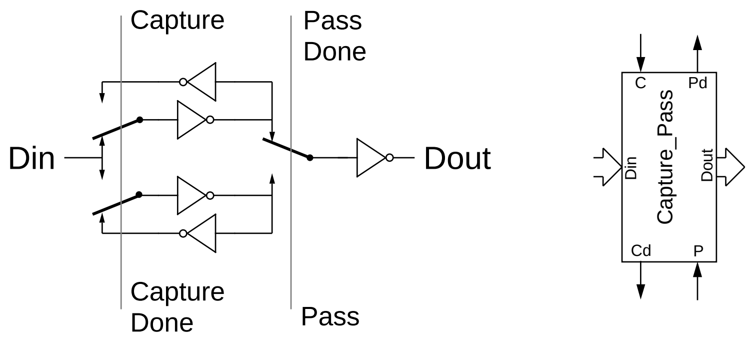


Figure 3-4: Event-controlled storage element

3.2 Micropipelines

Sutherland assigned the name Micropipeline to a simple form of event-driven elastic pipeline. The simplest Micropipeline is one in which there is no processing in between pipeline stages. Micropipelines cause data ordering to be maintained so that data exits the pipeline in the same order that it entered. This is referred to as a First In First Out (FIFO) queue.

3.2.1 A Micropipeline FIFO

Figure 3-5 shows **C-Gates** and **Capture-Pass** latches configured to form a 4 stage Micropipeline FIFO. The operation of the FIFO begins with the data being presented to the first **Capture-Pass** latch via *Din*. Initially all the event wires are Low and all the latches are transparent. The *Rin* signal arrives at the first **C-Gate** (as a Low to High transition) to indicate that the data is now valid and may be latched. The other input to this gate is pre-initialised, so that although no event has yet arrived from *Pd* the input of the **C-Gate** is primed and the *Rin* event propagates to the latch control circuits, closing the first latch. The "capture done" (*Cd*) control wire indicates when the latch has closed and this generates an acknowledge event on *Ain*. Once this operation is complete the data may be removed from the input to the FIFO. The "capture done" (*Cd*) event is also fed into a delay unit in the path to the next stage of the FIFO. This slows down the event signal thus giving the data time to propagate through the rest of the first latch stage and to arrive at the second stage, ensuring that the bundled-data delay constraint for the second stage is not violated.

The delayed event arrives at the primed **C-Gate** in stage two. The data is now safely set up at the input of the second stage and the latch closes after the event propagates through the **C-Gate**. Again, once closed, the stage signals back to its predecessor that the data is no longer needed. This causes the first stage to become transparent and primes its input **C-Gate** so that a subsequent *Rin* may propagate straight through to close the latch. An *Rin* event which arrives before the latch has opened is stalled at the **C-Gate** awaiting the latch empty (pass done - *Pd*) event.

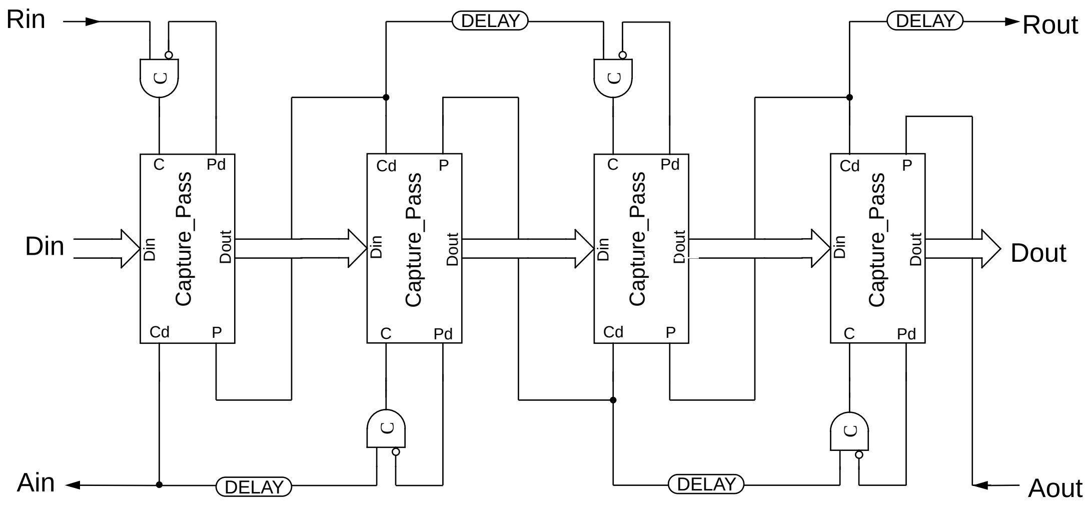


Figure 3-5: Simple Micropipeline FIFO

The control signals continue to ripple down the FIFO towards *Rout*, latching the data at each stage and releasing the previous stage as they progress. Eventually the data and its corresponding event arrive at the output and the environment is signalled on *Rout* to indicate that the data is now available on *Dout*.

If the data is not removed from the output and more data is added at the input, the incoming data progresses down the pipeline until a forward request reaches a non-empty stage. As described above, a full stage will be inhibited by the **C-Gate** from processing any further requests until it is first empty. This may eventually lead to all the FIFO stages becoming full, the FIFO will then remain back-logged until data is removed from its output.

3.2.2 Micropipelines with processing

The simple Micropipeline FIFO can be extended to include processing functions by the addition of logic interspersed between adjacent latch stages (figure 3-6). This operates in a similar manner to the empty FIFO with events rippling down the Micropipeline. The delay in the forward request propagation path must be increased to allow for the delay incurred by the data passing through the processing logic.

More complex structures, such as forking or merging pipelines, can be constructed with the aid of other library elements (i.e **SELECT**, **XOR** etc.).

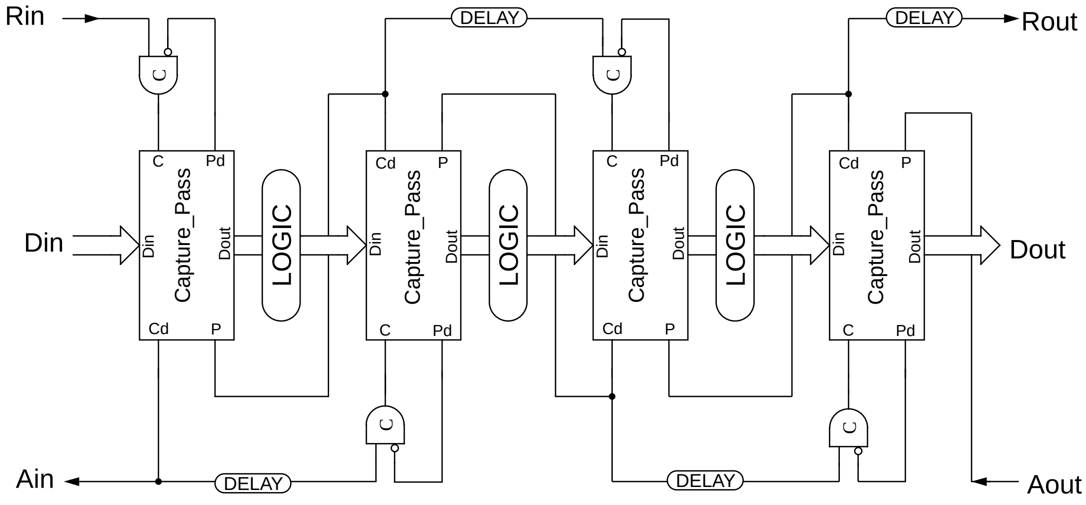


Figure 3-6: Processing Micropipeline

3.3 Event control module structures

The review of Micropipelines described so far is based upon the work presented by Sutherland in the Turing Award lecture. To construct a Micropipeline system an implementation of the basic event module library is required. This section describes the event module implementations chosen for the asynchronous microprocessor.

3.3.1 Exclusive OR gate

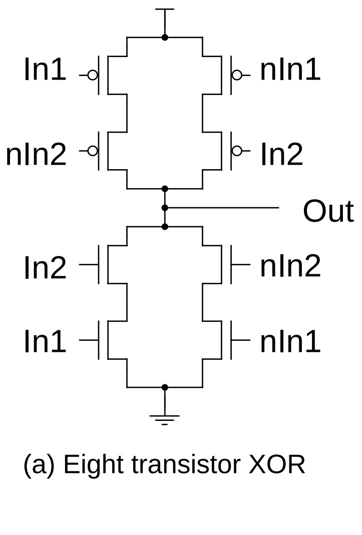
Figure 3-7 shows two standard implementations of an exclusive OR gate, (a) shows a standard eight transistor implementation and (b) a six transistor version [Shoj88]

The eight transistor **XOR** assumes that both the true and complement of both inputs are available; if this is not the case they must be generated locally at the cost of an inverter for each input (2 transistors). In the course of the design of the asynchronous ARM processor it was observed that in most cases at least one complement can be made available by the environment.

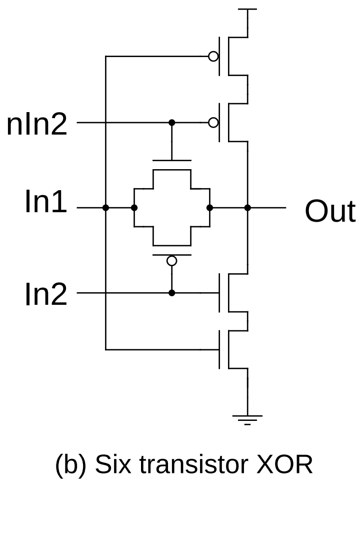
The six transistor **XOR** needs the complement of only one of its inputs. Initially this appears a more cost effective **XOR** in terms of the number of transistors; however the circuit suffers from a charge sharing problem. Consider the following case: Initially *In1* and *In2* are High. The transmission gate is turned off and the output is pulled low via the n-transistor stack. If *In2* goes Low (and *nIn2* goes High) then the n-transistor stack turns off and the transmission gate turns on. The transmission gate initially has a High at one end (*In1*) and a Low at the other (*Out*). The resulting charge-sharing causes a glitch to appear on the *In1* input. A glitch is effectively 2 transitions and would cause unexpected events to propagate through the control circuitry if it passed the switching threshold of gates connected to it. Analysis with SPICE [Nage73] revealed that the glitch could reach a voltage of 0.8 V. The circuit was discarded on two grounds:

- Every instance of the gate would need to be checked to ensure that the glitch did not cause problems with its immediate environment.
- Any glitches generated by other circuitry could be mistaken as **XOR** glitches instead of real errors.

The eight transistor **XOR** has therefore been used with additional local inverters to generate the complements of the inputs as required.



(a) Eight transistor XOR



(b) Six transistor XOR

Figure 3-7: Exclusive OR gates

3.3.2 Muller C-Gate

Various implementations of **C-Gates** have been proposed [Suth86]. The one adopted in the asynchronous ARM design is based upon the dynamic Muller C-element described by Sutherland. It was chosen in preference over the other designs on grounds of simplicity and reduced transistor count.

The dynamic Muller C-element can be made pseudo-static by the addition of a weak feedback inverter to maintain the state of the intermediate node (see figure 3-8). The strength of the feedback inverter can be made sufficiently low (e.g. $1/_{25}$th normal strength) that it incurs a negligible performance penalty on the gate overall.

The operation of the gate is quite simple. When both inputs are Low the p-transistors are turned on and the intermediate node (*i*) is pulled High thus forcing the output Low. When either *In1* or *In2* makes a transition to High the pull-up stack is turned off and *i* is floating; the state of the gate is retained however by the weak feedback inverter.

When both *In1* and *In2* have made transitions so that they are both High the n-transistors are turned on causing the intermediate node to be pulled Low and hence the output High. This sequence of operations is repeated for High to Low transitions of the input.

All event modules with internal state retention are designed so that they can be initialised into a known state. The choice of initial state is arbitrary but to simplify circuit design all event modules were defined to reset Low. The **C-Gate** implementation therefore contains initialisation circuitry to enable the output to be reset Low regardless of the state of the inputs. The reset signal (*Cdn - Clear down*) is active Low. Early designs of this gate used only a pull-up p-transistor to reset the gate. This required the co-operation of the environment to ensure that the inputs were not both High during initialisation (hence turning on the n-transistor pull-down stack in opposition to the reset pull-up). Practical circuits often contain loops where it was not possible to ensure that the input preconditions for this type of reset could be met, so the full safe reset was implemented.

During reset the inputs of the **C-Gate** are forced Low by the environment (all other event modules in the environment should be forcing their outputs low). Often it is necessary to prime one of the inputs, for example in the simple Micropipeline shown in figure 3-5. This can be done by simply placing an inverter on the input to be primed. The initial Low value

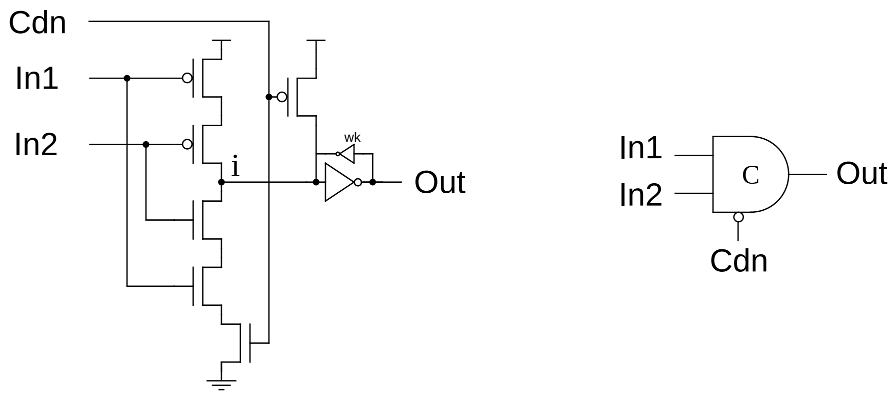


Figure 3-8: Pseudo-static Muller C-Gate

on the primed input presents a High value internally to the **C-Gate** input transistor stack, hence activating the corresponding n-transistor. In this state the n-transistor stack in the **C-Gate** is half turned on so it requires only a single event after reset (Low to High transition) on the non-inverting input for an event to appear on the **C-Gate** output.

In conventional CMOS gates the switching threshold is a function of the relative strengths of the pull-up and pull-down stacks; however in this **C-Gate** implementation this is not the case because the input stack does not switch directly from pulling-up to pulling-down but instead the sequence of operation is from pulling-up to high impedance to pulling-down. Therefore there is no overlap between pulling-up and pulling-down which gives the normal balanced threshold [Eshr89, chapter 2]. The threshold of the **C-Gate** is the threshold of the n-transistor or p-transistor which is much less than that of a standard static CMOS gate (nominally half the supply voltage assuming correctly sized transistors). The pseudo-static **C-Gate** is therefore a low threshold device; it turns on early. This must be taken into account when designing with this element.

3.3.3 Transparent latch

A transparent latch (**T-Latch**) can be implemented using a structure very similar to that of a **C-Gate** (figure 3-9). If the data is presented on *In* when enable (*En*) is High it will propagate to the output. When enable is Low the input transistor stacks are turned off and the state is sustained by the weak feedback inverter. The latch is also provided with reset circuitry similar to the **C-Gate.**

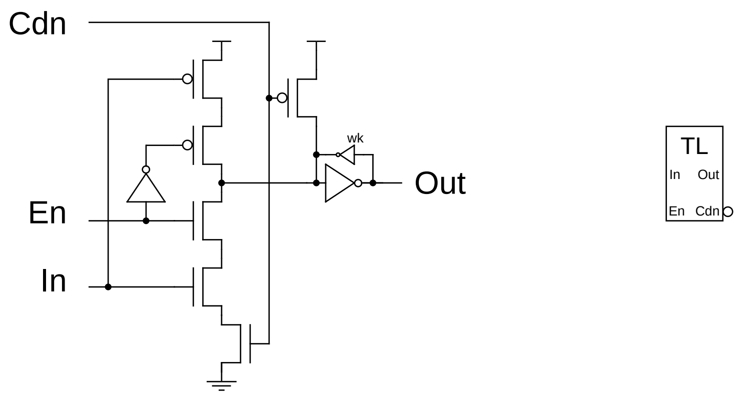


Figure 3-9: Transparent latch

3.3.4 SELECT block

The **SELECT** block can be implemented using transparent latches and **XOR** gates as shown in figure 3-10. The operation of the circuit is as follows: After initialisation the two latch outputs will be Low and the event input (*In*) will also be Low (the environment must ensure this). The *Sel* input is then set to determine which path the next event will take. If, for example, the *Sel* input is asserted High, the event will be steered to the *True* output.

With the *Sel* input High, the lower latch will be transparent and the upper latch will be opaque. When an event arrives on *In* it propagates through both **XOR** gates and arrives at

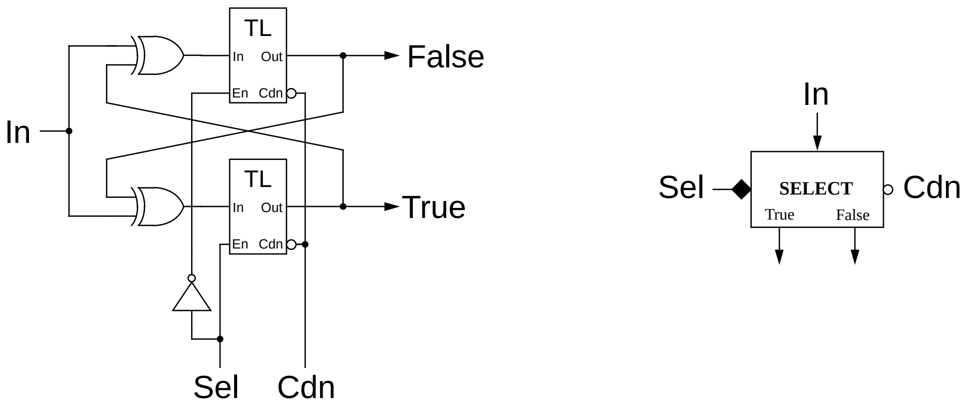


Figure 3-10: SELECT block

the inputs of both **T-Latches**. The upper latch is opaque so the event propagates no further; however the lower latch is transparent and allows the event to pass through to the *True* output. The output event is fed back via the upper **XOR** to cancel the request waiting at the input of the upper latch thus preventing an erroneous transmission of an event to the *False* output on a *Sel* change.

With *Sel* Low, events are steered to the *False* output in a similar fashion.

The environment must ensure that the *Sel* input is defined a sufficient time prior to an event arriving and is stable for enough time afterwards to meet the setup and hold times of the latches.

3.3.5 Decision-Wait element

The **Decision-Wait** element [Kell74] is not one of Sutherland's library elements, but it is a popular library element with other asynchronous design styles [Jose90] and can be used to construct a **CALL** block.

The **Decision-Wait** block causes an event on *Fire* to rendezvous with an event on either *a1* or *a2* (but not both) giving an event on the corresponding output (*z1* or *z2*).

The implementation of a **Decision-Wait** is shown in figure 3-11. The structure and operation of the gate is similar to the **SELECT** block. The detailed operation proceeds as follows: an event arrives on either *a1* or *a2.* For example, assume that an event arrives on a1. The arrival of this event primes the upper **C-Gate.** When an event arrives on *Fire* this propagates through the **XOR**s to the inputs of both **C-Gates**. The lower **C-Gate** is stalled waiting for an input on *a2* and so no further action results from this gate. However the upper **C-Gate** has now had an event on both inputs and so propagates an event to its output and hence *z1.* This output is fed back via the **XOR** to the lower **C-Gate** to cancel the pending *Fire* event on its input. Operation is similar for an event on *a2* except the subsequent output event will be on *z2.*

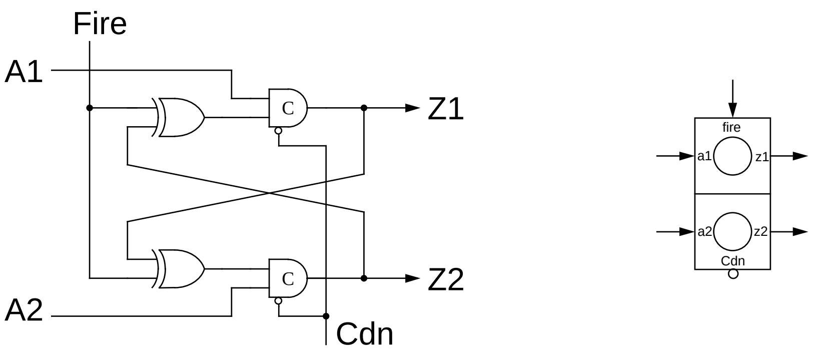


Figure 3-11: Decision-Wait element

3.3.6 CALL block

The **CALL** block is constructed from a **Decision-Wait** element and a single **XOR** as shown in figure 3-12. The operation begins with an input request on either *R1* or *R2*. This event passes through the **XOR** thus forming the request out (*R*). At the same time the corresponding *a1* or *a2* input of the **Decision-Wait** element is primed. When the acknowledge returns from the sub-circuit (on *D*) it will rendezvous with the primed input of the **Decision-Wait**, causing an event to be generated on the acknowledge (*D1*/*D2*) back to the correct calling circuit.

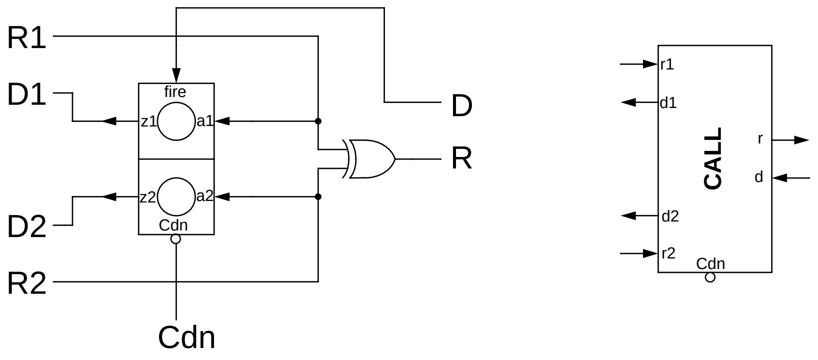


Figure 3-12: Call block

3.3.7 TOGGLE

The **TOGGLE** proved to be the most difficult circuit element to implement safely. Figure 3-13 shows a high-level view of a **TOGGLE** circuit. The operation of the circuit is to allow a transition to circulate around a loop under the control of two transparent latches (**TL**). Every time there is an event on the input (*In*) the latches allow the transition to propagate one position round the loop. This is done by opening one latch while at the same time closing the other. Care must be taken to ensure that both latches do not remain open together during the change over, otherwise the transition may propagate through two positions and

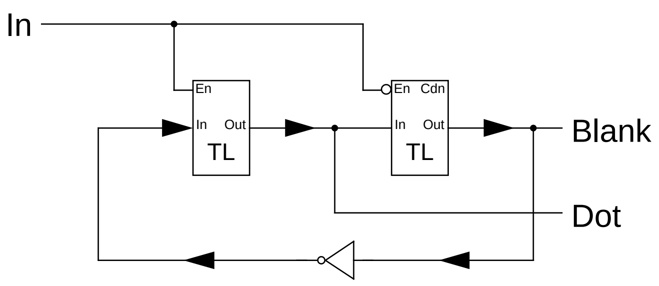


Figure 3-13: TOGGLE high-level operation

generate a spurious transition on one of the outputs. This is particularly troublesome when the latches are made from low threshold devices such as transmission gates, where the latches open early and close late compared to standard gates. This caused “race-through” problems with early designs which could be made to operate reliably only by controlling the latches with non-overlapping clock generators.

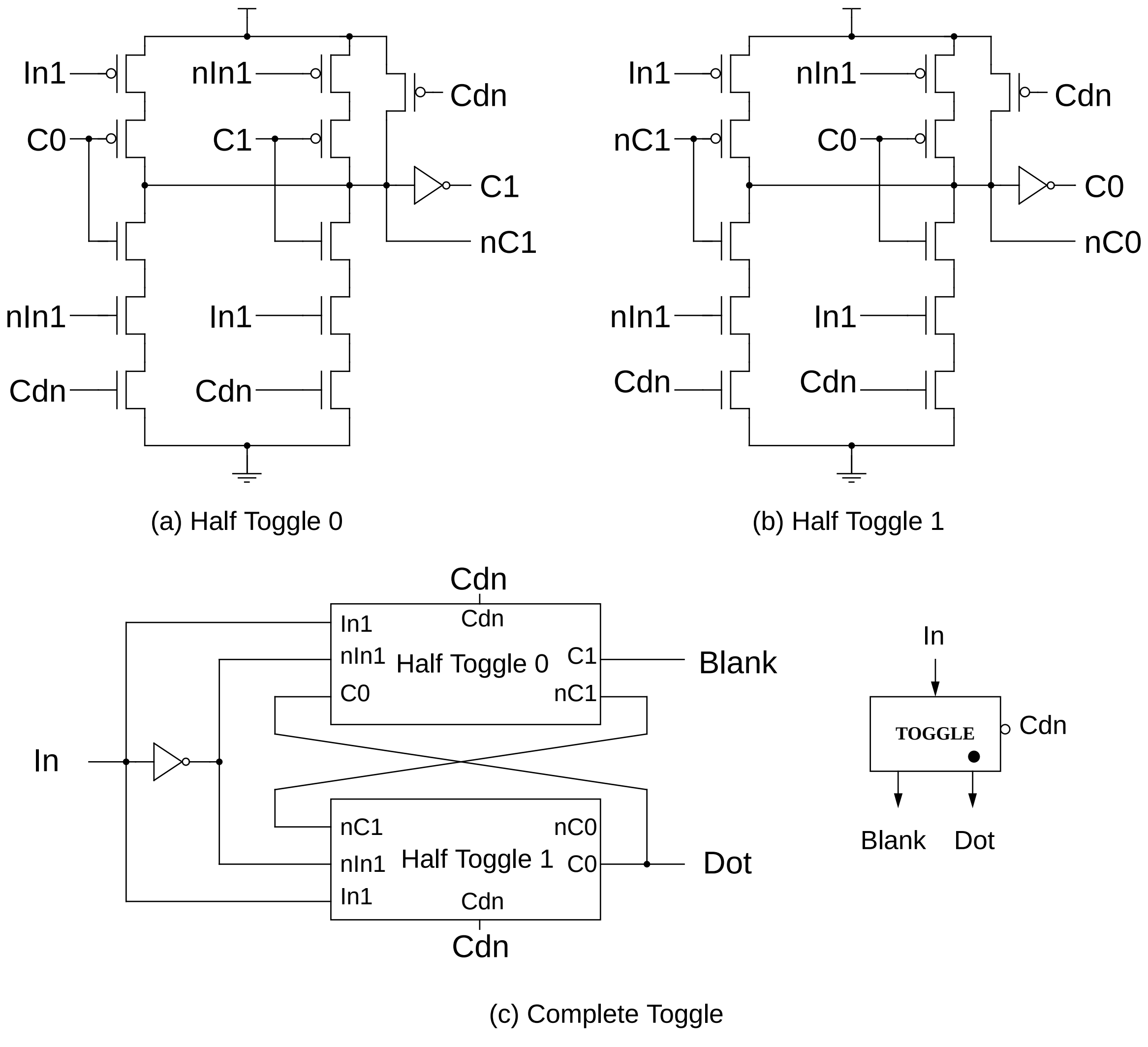


Figure 3-14: TOGGLE element

The **TOGGLE** circuit eventually chosen was based upon a design by Jay Yantchev of Oxford University PRG [Yant92]. This design was derived by speed-independent decomposition of the **TOGGLE** [Jose92] specification using an algebraic approach (S-I Algebra). The complete circuit is shown in figure 3-14. The two half toggles are transparent latches similar to previous designs [Eshr89, fig 5.51(a)] and the overall operation is as described above.

The speed-independent derivation assumes that both *In* and the complement *nIn* switch simultaneously. In an implementation this is never actually the case, and SPICE simulation showed that if *In* and *nIn* were sufficiently skewed the circuit could indeed be made to fail. For this reason in the silicon implementation the complement *nIn* is generated locally within the same cell so that the delay can be carefully controlled.

After adding the local *nIn* inverter and reset circuitry to the initial design, the resulting circuit was converted to silicon layout. The circuit parameters were then extracted and analysed with SPICE to examine the behaviour at all process corners and at a variety of temperature and voltage combinations. This demonstrated that the circuit functioned correctly under all conditions and the inverter delay to complement the input was within the limits required to prevent latch breakthrough.

Other **TOGGLE** designs were considered including a circuit based on a TTL style D-type flip-flop implemented in NAND gate technology [TTL85]. This is shown in figure 3-15 and was demonstrated to be hazard free [Edwa92]. This was not adopted because of the increased complexity and half strength output drive compared with the Yantchev circuit (NAND gate compared to inverter drive).

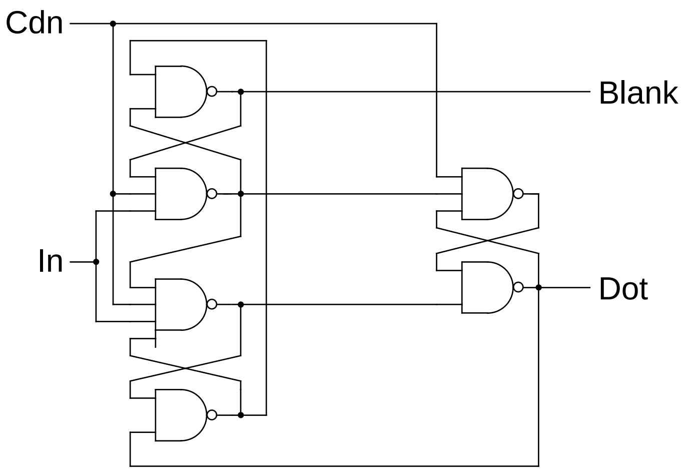


Figure 3-15: NAND gate TOGGLE

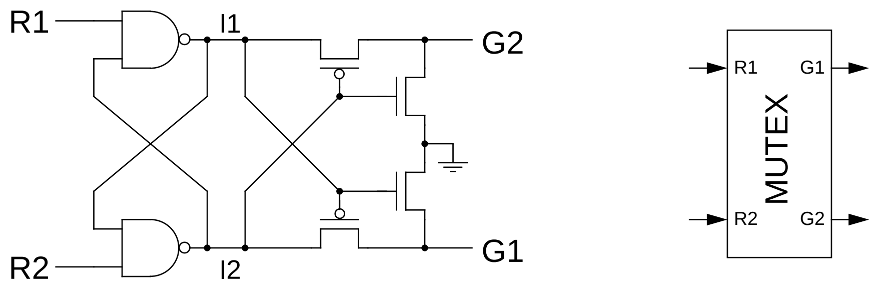
3.3.8 ARBITER

The mutual exclusion circuit shown in figure 3-16(a) is based on a CMOS implementation of a well-tried NMOS circuit [Mead80, figure 7.25]. The circuit comprises a pair of cross-coupled NAND gates configured as an R-S flip-flop. The output of each NAND gate is connected to the input of one of the output inverters and acts as the power supply for the other. If an input is activated by going High the corresponding internal node goes Low. For example if *R1* goes High the internal node *I1* goes Low. Once the internal node is Low this prevents subsequent events on the other input (*R2*) from having any effect. The other internal node, *I2*, remains High. As *I2* also acts as the power supply for the lower inverter, the output *G1* of this inverter is pulled High indicating that arbitration is complete.

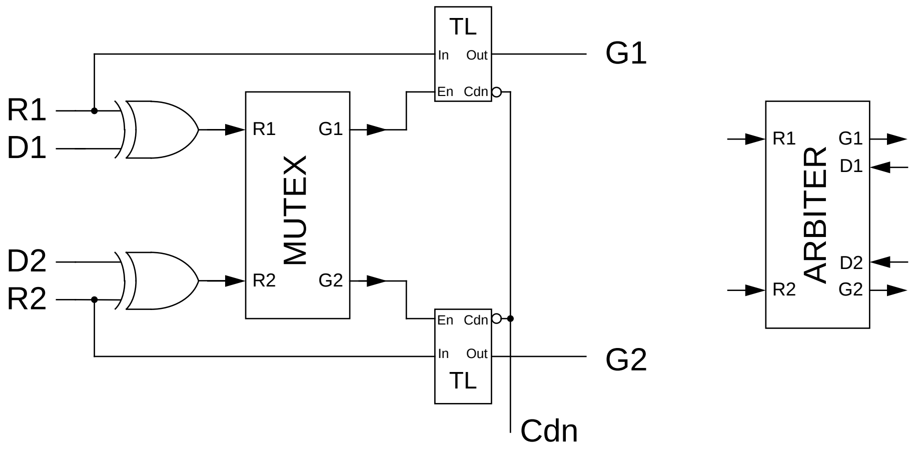
If both inputs go high at the same time the R-S flip-flop may go into a metastable state where both internal nodes remain at an intermediate, (non-digital level) for an arbitrary amount of time. With intermediate levels on the internal nodes the output inverters are unable to turn on and pull either of the outputs High, so the outputs remain Low since the n-transistors are still partially turned on.

Eventually the R-S flip-flop will exit from the metastable state and one of the internal nodes will settle Low, the other High. When the difference between the internal nodes is more than a p-transistor threshold one of the inverters will begin to turn on and start pulling its output High.

The mutual exclusion circuit is released by removing the input request, i.e. *R1* returns Low. This causes the internal node *I1* to return High and allows any pending requests on *R2* to be serviced.



(a) Four-phase mutual exclusion circuit



(b) Two-phase Micropipeline arbiter

Figure 3-16: ARBITER

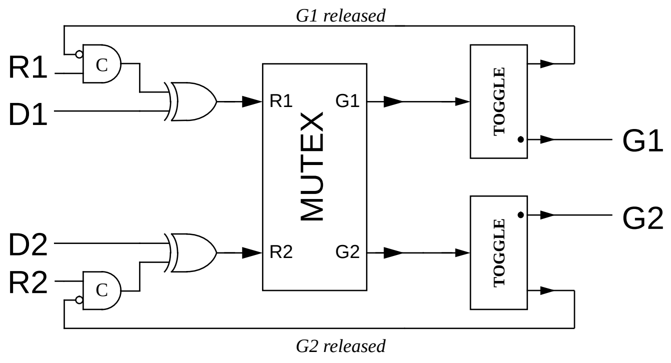


Figure 3-17: Robust two-phase arbiter

The circuit described above is a four-phase circuit, in that the input must return to zero when the operation is complete. Figure 3-16(b) shows how this mutual exclusion circuit can be used to build a two-phase **ARBITER** suitable for Micropipeline control circuits. An incoming two-phase event is converted to four-phase by the **XOR** gate on the input. This ensures that the mutual exclusion circuit always receives a Low to High transition to activate it. After arbitration is complete one of the transparent latches is enabled to allow the successful incoming two-phase event to propagate to the output. When the sub-circuit issues an acknowledge in the form of an event on either the *D1* or *D2* inputs, the corresponding input of the mutual exclusion returns Low and hence releases the **ARBITER** and closes the corresponding transparent latch.

Figure 3-17 shows a more robust variation of a two-phase **ARBITER** implementation which allows simultaneous events on the *r* and *d* inputs. The extra **C-Gate**s and **TOGGLE**s ensure that the four-phase mutual exclusion circuit is released before allowing another request on the same input.

3.3.9 Capture-Pass latch

The implementation of the **Capture-Pass** latch is shown in figure 3-18. The operation of the latch is as follows:

After reset, the capture (*C*) and pass (*P*) signals are configured so that the input is connected to the upper branch and the output is also connected to the same branch. In this state the latch is transparent.

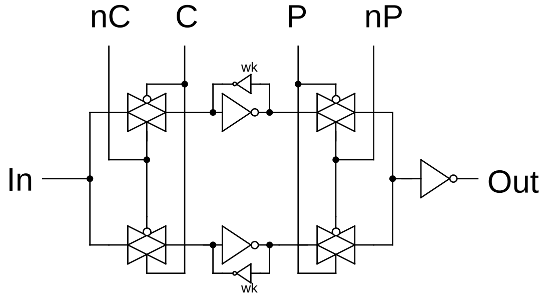


Figure 3-18: Capture-Pass implementation

Transitions on the capture control wires (*C* & *nC*) cause the upper input transmission gate to close and the lower one to open. The state of the upper branch is sustained by the weak feedback inverter and is still visible at the output. This configuration remains until the arrival of a transition on the pass control wires (*P* & *nP*). This causes the upper output transmission gate to close and the lower one to open. The output is now fed from the lower branch, which is already connected to the input, so the latch is again transparent. The cycle repeats with the input and output being switched back to the upper branch on the next transitions of the capture and pass control wires.

3.3.10 Cell layout

The cells were implemented as standard-cells[1] so that they could be used in conjunction with the ARM Ltd. standard logic element library (inverters, NAND gates etc.) to form layout automatically compiled by conventional standard-cell place and route software [Comp91]. The transistor sizes in the asynchronous cells were chosen to be compatible with the ARM library and are integer multiples of the single inverter transistor sizes. The target process was a 1.2 micron, twin-tub, double-layer metal process (the VLSI Technology CMN12 process [VLSI91]).

Figure 3-19(a) shows an example of the **C-Gate** standard-cell layout. The cells have a fixed power supply pitch (*Vdd* & *Vss*) running horizontally on metal 1. Connections to the cell's inputs and outputs are made vertically on the metal 2 layer. The weak feedback inverter of the **C-Gate** can clearly be seen on the right of the cell with its characteristic "C" shaped pull-down n-transistor. The transistor is so shaped to obtain minimum gate width and to maximise gate length so giving the desired weak pull-down attribute.

Complex cells can be constructed from simpler cells with the interconnect wiring external to the power supplies (effectively in the routing channel). Figure 3-19(b) shows the **CALL** block which is made up of 2 **C-Gates** and 3 **XOR** gates.

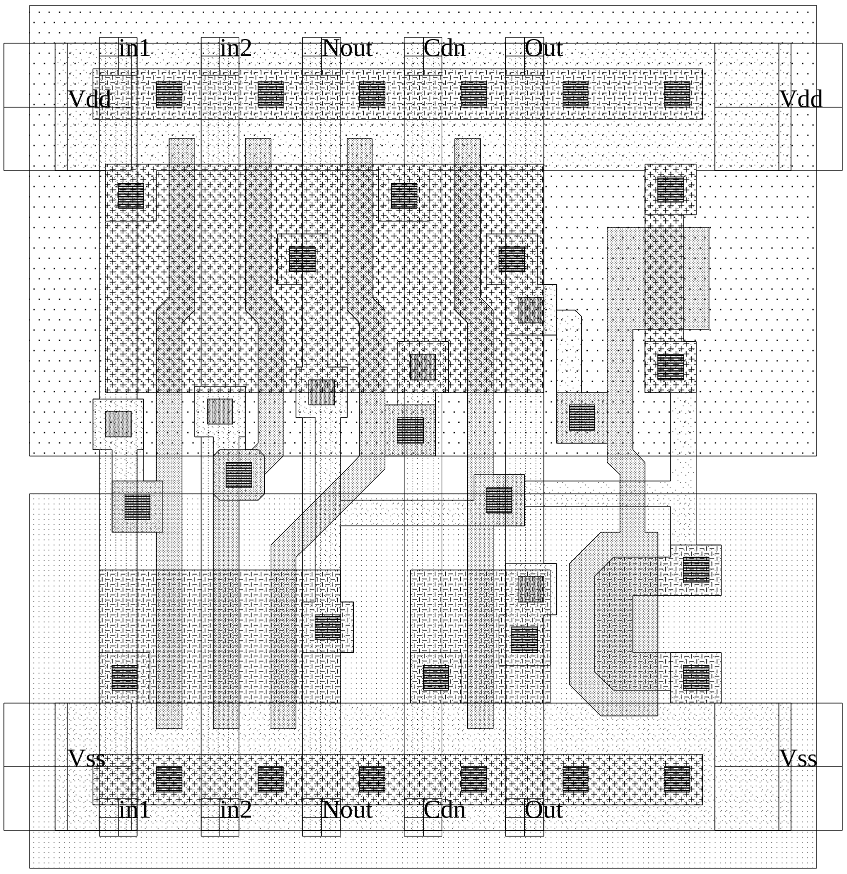
The layout cells were extracted and their operation analysed with SPICE under varying load conditions. The resulting timing characteristics can be found in appendix A.

3.3.11 Implementation costs

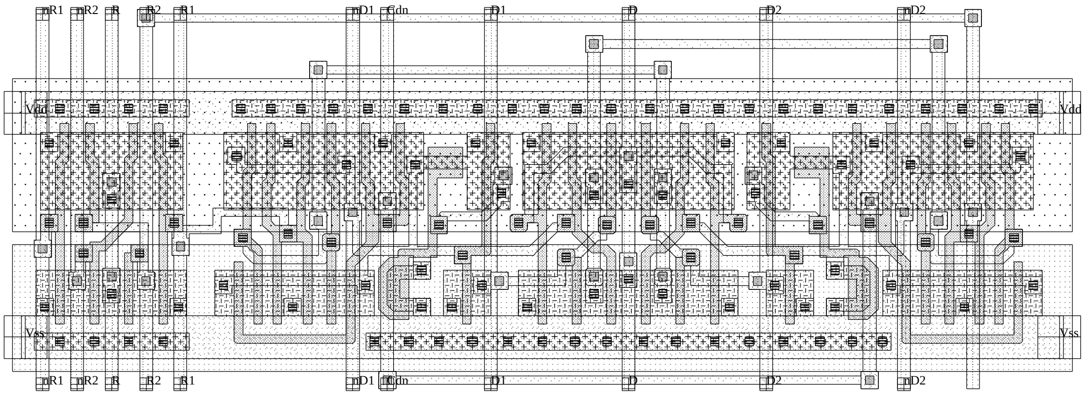
Figure 3-20 shows a plot of the relative sizes of the asynchronous cells taken from the layout editor of the Compass Design Automation VLSI design tools [Comp91a]. There is a single strength inverter (**Inv1**) on the right to act as a scale reference point.

In principle it is possible to extend most of the asynchronous elements so that they have more inputs/outputs. An example of this would be to extend the select block to steer an incoming event to, say, one of 4 outputs. This can be done by having two select lines, four **T-Latches** and four 4 input **XOR** gates. These are wired so that each **T-Latch** output is connected to the **XOR** gates on the inputs of the other 3 **T-Latches**. The overall operation is as described previously but this time the event fed back cancels the event on the other three **T-Latches**. Figure 3-20 shows that although in theory this may be a feasible idea, in practice a **SELECT4** built out of primitive elements is more than 2.5 times bigger than a

1. Standard-cells are VLSI cells that have a fixed power rail pitch and may be butted together to form rows of cells that may be wired together by automatic routing tools.



(a) Layout of a Muller C-Gate with reset



(b) Layout of a CALL block

Figure 3-19: Examples of cell layout

SELECT4 built out of three **SELECT2** elements. The extra size can be attributed to the more complex **XOR** gates and the interconnect wiring required.

3.4 Micropipeline implementation

Sutherland described two principle implementations of a Micropipeline stage, one based upon a **Capture-Pass** latch as shown previously in figure 3-5 and the other based on transparent latches as shown in figure 3-21.

The example shown here is a 2-bit Micropipeline stage. An **XOR** and a **TOGGLE** element are needed to translate the input transition signalling into the correct form for controlling the level-sensitive enable of the transparent latches. The latches have active Low enables, i.e. they are transparent when enable is Low. Initially *C*, *P*, *Cd* and *Pd* are all Low, meaning both latches are transparent.

When a capture event occurs on *C*, this passes through the **XOR** and forms a Low to High transition on the enable of the latches, thus causing them to close. The same enable signal is sensed by the **TOGGLE** to indicate that the latches have closed and a subsequent *Cd* event is generated on the *Dot* output. The **TOGGLE** is connected to the same wire as the latch enables so that if the number of data bits is large and the enable signal has slow edges (caused by the large capacitive load) then the **TOGGLE** element also sees the slow edge and automatically compensates for it by waiting before switching until the enable signal has reached the threshold level (irrespective of the time taken to reach it).

When the stage is to return to a transparent state an event arrives on *P* causing a High to Low transition on the latch enable, causing the latches to open. This is again monitored by

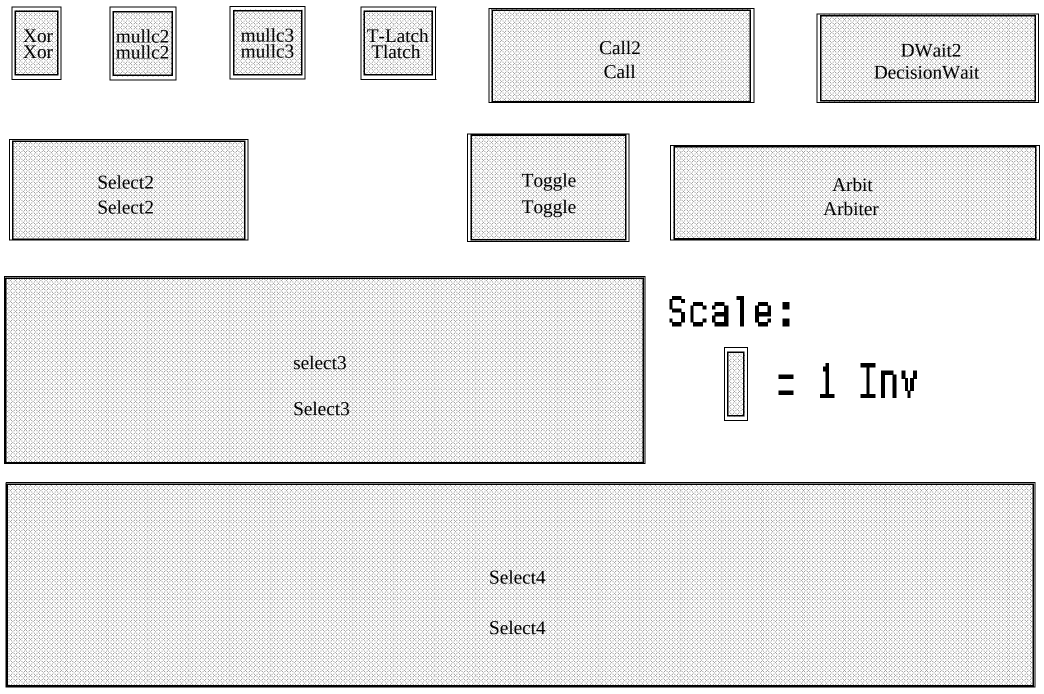


Figure 3-20: Cell silicon area comparison

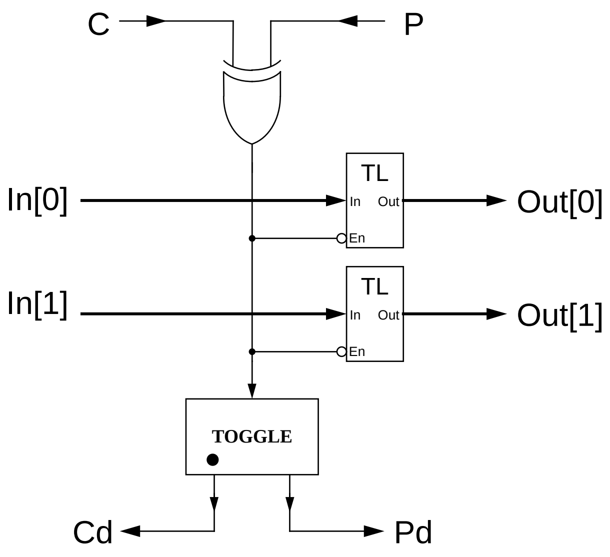


Figure 3-21: Simple transparent latch Micropipeline stage

the **TOGGLE** and an event is generated on the *Blank* output (*Pd*) to indicate when the latches are open.

At a higher level, the **XOR** can be viewed as merging the Capture (*C*) and Pass (*P*) event streams and the **TOGGLE** as separating them again into Capture done (*Cd*) and Pass done (*Pd*); together they form a two- to four-phase interface.

Two practical implementations of Micropipeline stages will be described and used as the basis of a comparison between the relative costs and performances of the two styles.

3.4.1 The T-Latch Micropipeline

The **T-Latch** design described earlier (see figure 3-9) is used in the control circuits and contains reset logic. This facility is redundant for datapath latches because they are forced transparent during initialisation. It is also possible to simplify the design by using a transmission gate as shown in figure 3-22. These latches are used for data storing of a bundle so any transistor saving per latch is multiplied by the number of bits in the bundle.

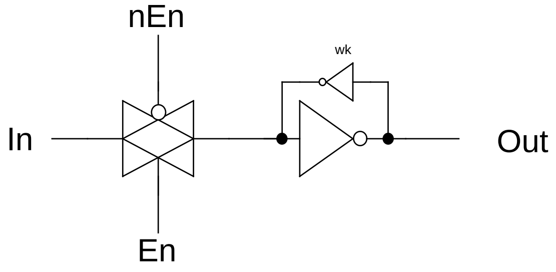


Figure 3-22: Simple transmission gate latch

The transmission gate on the input of the latches requires both the true and complement of the enable signal (*En* & *nEn*) to operate. Instead of providing the circuitry to generate the complement locally in each bit, the complement is generated by the control circuit and supplied to all the latches. This further reduces the transistor cost per bit.

In the simple circuit shown in figure 3-21, the **TOGGLE** is used to sense the level of the latch enable to compensate for enable line loading. A consequence of using a transmission gate latch is that there are now two enable signals (*En* & *nEn*) both of which must be sensed to ensure they have made the required transitions. This can be achieved by causing the two enable signals to rendezvous at a **C-Gate** before propagating to the **TOGGLE**.

Figure 3-23 shows the detailed control logic required to implement a Micropipeline based on transparent latches. The **T-Latches** (not shown) are connected to the *En* and *nEn* signals. The figure shows the buffers to drive true and complement enable lines and the **C-Gate** to synchronize the two enable lines.The small subscript on each buffer inverter indicates the drive strength as a multiple of single inverter capability. The control circuit shown was designed to drive a 32-bit data bundle.

The overall operation begins when an *Rin* event arrives at the input **C-Gate**. If the latch is already transparent (indicated by the value fed back from the **TOGGLE** *Blank* output) then the event propagates through the **C-Gate** and **XOR** and causes the latch to close. When both the true and complement enables have made a transition to the closed value the second **C-Gate** detects this and an event is forwarded to the **TOGGLE**. The *Dot* output of the **TOGGLE** indicates that the latch is now safely closed and this can be used to signal to the previous stage that the input value is no longer needed (*Ain*). It also forms the request to the next stage (*Rout*) to indicate that the data is now ready for further processing.

The stage is emptied when a returning acknowledge event (*Aout*) arrives via the **XOR** and opens the latches. This is detected by the **C-Gate** and **TOGGLE** and the resulting event on the *Blank* output of the **TOGGLE** is used to prime the input **C-Gate** ready for the next *Rin*.

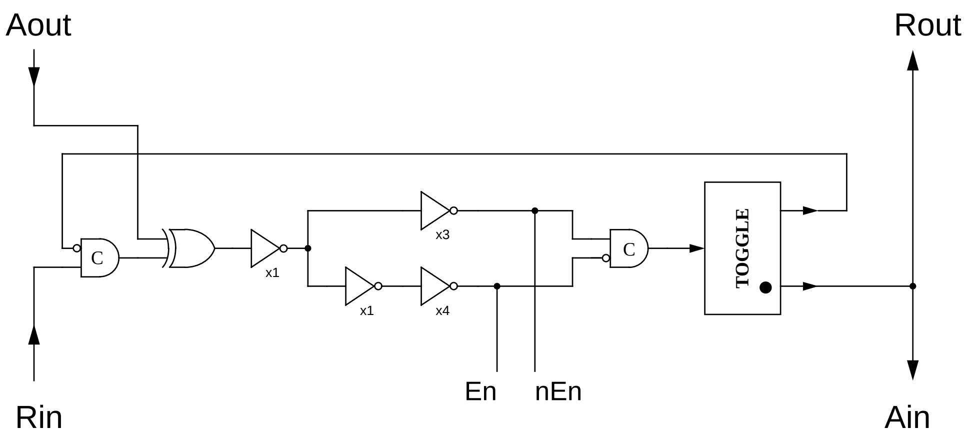


Figure 3-23: Transparent latch Micropipeline control

3.4.2 The Capture-Pass Micropipeline

Figure 3-24 shows an implementation of the control structure required for a **Capture-Pass** style Micropipeline. The “capture” and “pass” control wires of the data latch (see figure 3-18) would be connected to (*C & nC*) and (*P & nP*) respectively.

The operation begins with an input request (*Rin*) which can propagate through the input **C-Gate** when the data latch is empty. The signal is then buffered to drive the true and complement of the data latch capture control wires. To ensure the true and complement of the capture signal have made their transition, they are forced to rendezvous at the second **C-Gate** before an input acknowledge (*Ain*) and an output request (*Rout*) are generated.

The data latches remain closed until an acknowledge (*Aout*) is received from the next stage in the Micropipeline. This signal is buffered and used to operate the pass signals of the data latches and causes them to return to transparent. Again both control wires are synchronised with a **C-Gate** before the input rendezvous is primed ready to accept the next *Rin*.

3.4.3 Capture-Pass versus transparent latch area considerations

Figure 3-25 shows the silicon implementation of an 8-bit, 3 stage Micropipeline and the corresponding control for both **T-Latch** and **Capture-Pass** styles. The control is automatically compiled standard-cells and the data part is hand composed custom layout.

The **Capture-Pass** latches shown are a silicon implementation of figure 3-18. Their size is slightly more than double that of a **T-Latch**. However, the **Capture-Pass** latch has not been as highly optimized as the **T-Latch** so a size ratio of 2:1 is a fairer estimate of the relative area cost of a **Capture-Pass** latch compared with a **T-Latch**.

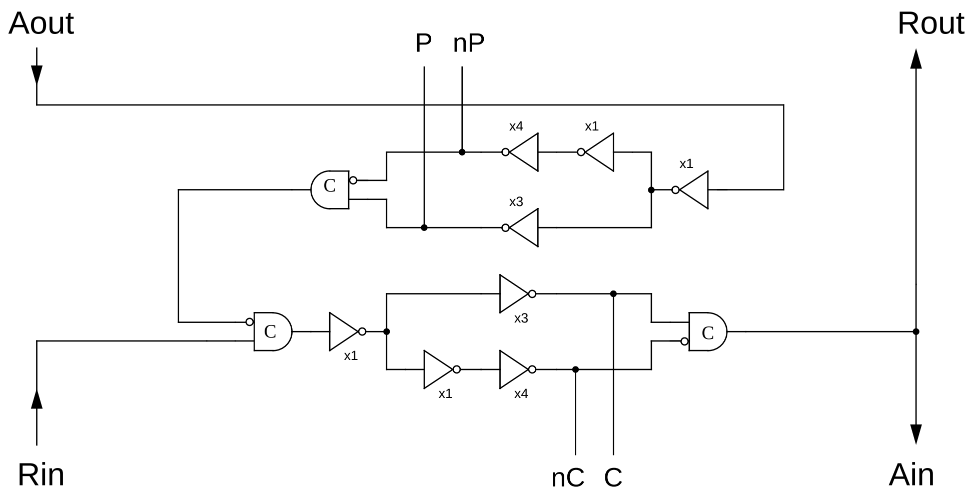


Figure 3-24: Capture-Pass Micropipeline control

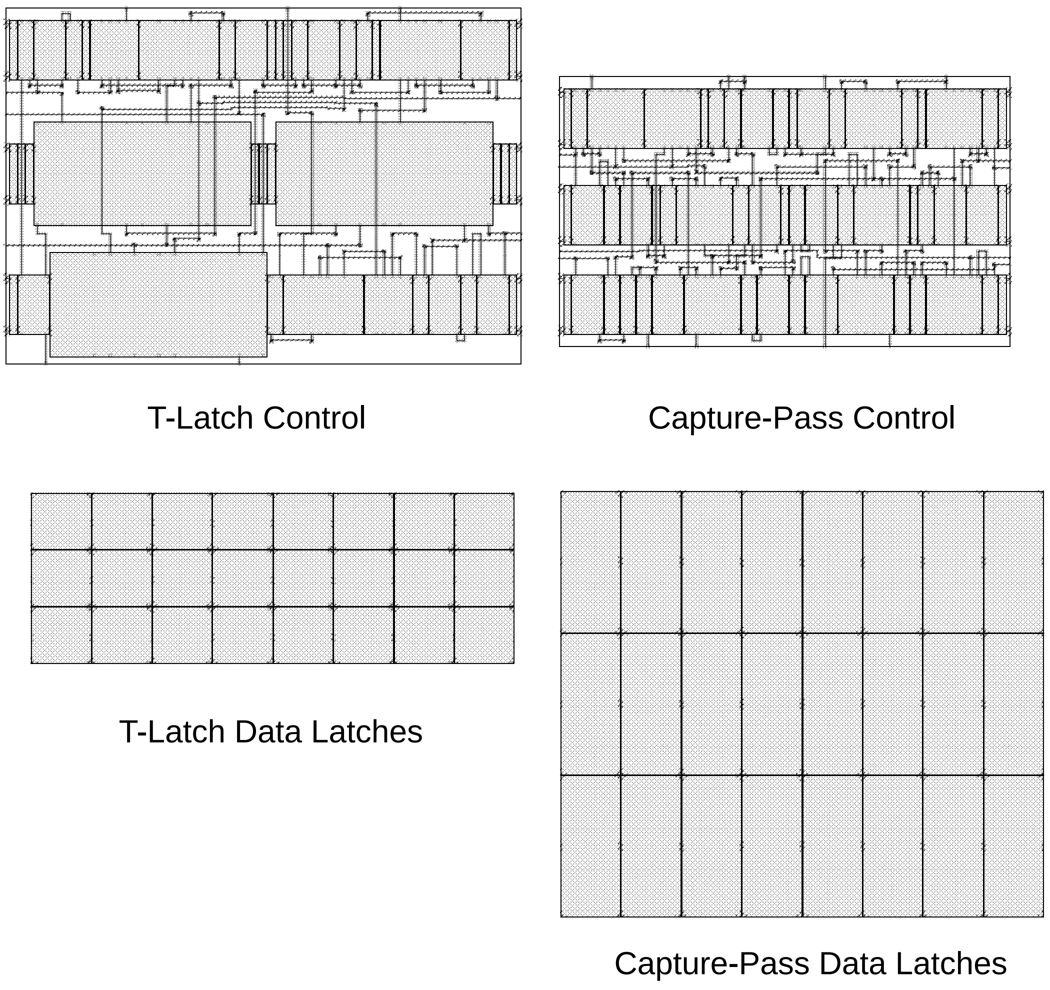


Figure 3-25: Relative size of Capture-Pass v T-Latch Micropipeline

By measuring the area of the control structures and using a 2:1 size ratio on the data part it is possible to work out which style is more area efficient for 8-bit data bundles (if **T-Latch** is area *D* then **Capture-Pass** is area 2x*D*).

Table 1:

T-Latch Area	Capture-Pass Area
TControl + Data	CControl + 2 x Data
$164352 + 77875\ \lambda^2$	$119616 + 155750\ \lambda^2$
$242227\ \lambda^2$	$275366\ \lambda^2$

where *TControl* and *CControl* are the areas of the control logic for the two Micropipeline stages and *Data* the area per data bit.

Note: for the VLSI CMN12 process λ=0.6 Microns

This shows that **T-Latches** are more area efficient for this 8-bit example.

It is clear from figure 3-25 that the control overhead is smaller for the **Capture-Pass** implementation and this is reinforced by the figures given in the calculation shown above. It can therefore be expected that the **Capture-Pass** style will become more efficient as the size of the data bundle decreases. The cross-over point for **Capture-Pass** becoming more efficient can be calculated as follows:

Table 2:

T-Latch Area	>	Capture-Pass Area
TControl + N x Databit	>	CControl + N x 2 x Databit
$\frac{TControl - CControl}{Databit}$	>	N
4.5	>	N

where *N* is the number of bits and *Databit* the area cost per data bit.

This demonstrates that **Capture-Pass** latches are more area efficient for smaller data bundles (N < 5) and **T-Latches** are more area efficient for large bundles. The target application contains mainly 32-bit data bundles, in this case the **T-Latch** implementation is 36% smaller than the **Capture-Pass** latch. As the data width increases the size of the data part becomes the dominant factor of the area with the control becoming negligible. Therefore in the limit, as data width increases, the **T-Latches** would be approximately 50% smaller than the corresponding **Capture-Pass** implementation as this reflects the relative sizes of the two data latch styles.

3.4.4 Micropipeline stage performance

The performance of a Micropipeline can be considered in two ways. The first is the time taken to propagate through the stages, this is referred to as the *latency*. In figure 3-23 this would be the time from *Rin* to *Rout*. The second performance factor is how soon the stages can accept the next value, referred to as the *cycle-time*. The number of items that can be processed in a unit time is known as the *throughput* or bandwidth.

Using the figures in appendix A and the SPICE characteristics of the ARM Ltd. standard cells [ARM91a], the latency of a 32-bit **T-Latch** Micropipeline (i.e. the time from *Rin* to *Rout*) can be calculated.

The time from Rin to Rout can be broken down as follows:

C-Gate	+ **XOR**	+ Buffers	+ **C-Gate**	+ **TOGGLE**
2.16	+ 0.97	+ 4.3	+ 2.16	+ 2.70 = 12.29 nS

A similar analysis of a **Capture-Pass** style Micropipeline would give:

C-Gate	+ Buffers	+ **C-Gate**	
2.16	+ 5.13	+ 2.16	= 9.45 nS

The extra delay for the buffer circuit in the latter is attributed to the extra capacitive load on the **Capture-Pass** lines (twice that of a simple **T-Latch**).

The minimum cycle time is achieved by directly connecting *Rout* to a similar Micropipeline stage without intervening logic. The delay before the next value can be processed in the **T-Latch** design is when the **TOGGLE** signals back to the input **C-Gate** that the latches are now open i.e after:

Rin->Rout	+ *Rout->Aout*+	**XOR**	+ Buffers	+ **C-Gate** + **TOGGLE**
12.29	+ 12.29	+ 0.97	+ 4.3	+ 2.16+ 2.70 = 34.71 nS

Again doing a similar calculation for **Capture-Pass** style stages the cycle time is:

Rin->Rout	+ *Rout->Aout*+	Buffers	+ **C-Gate**	
9.45	+ 9.45	+ 5.13	+ 2.16	= 26.19 nS

From these figures it can be observed that the latency of a **Capture-Pass** stage is 23% less than the **T-Latch** stage and also the cycle time is approximately 25% less.

Taking these figures at face value indicates a 25% performance gain by using the **Capture-Pass** latches; however a fairer comparison can be made by including other design issues in the calculation.

The increased time taken for *Rin* to *Rout* in the **T-Latch** Micropipeline can be viewed as an increased "delay" before the *Rout* reaches the next stage. This corresponds well with Sutherland's model of a "delay element" as shown in figure 3-5. This extra delay from *Rin* to the latches closing increases the bundled-data delay margin of the data at the **T-Latches,** and the delay from the latch closing to the request out (*Rout*) (and acknowledge back (*Ain*)) ensures the **T-Latch** hold time is met and this also increases the bundled-data delay margin of the *Rin* to data in the following stage.

If the simple Micropipeline is extended, so that processing logic is interspersed between the stages then the extra margin of the **T-Latch** can be used to contribute to the matched delay required for logic. Therefore the delay in the control of the latches can be hidden to some extent by the processing logic.

It is possible to improve the latency of the **T-Latch** style design by forwarding the request to the next stage earlier. To avoid metastability problems the data must be stable at the output of the data latches before they are closed. Therefore the signal that closes the latches may be used to form an early request out (*Rout*) without waiting for the latches to close. The input cannot be acknowledged until the latches are closed so *Ain* is connected as before. Figure 3-26 shows the control circuitry with the "fast-forward" connection. The overall operation is the same as before except that *Rout* is derived from a different point in the circuit.

The latency of the circuit is substantially reduced to approximately 2.5 nS (i.e 1 **C-Gate** with increased capacitive load) and the cycle time is also reduced because the *Rout->Aout* part of the cycle begins earlier. The bundled-data delay safety margin is reduced so care must be taken when using this control circuit.

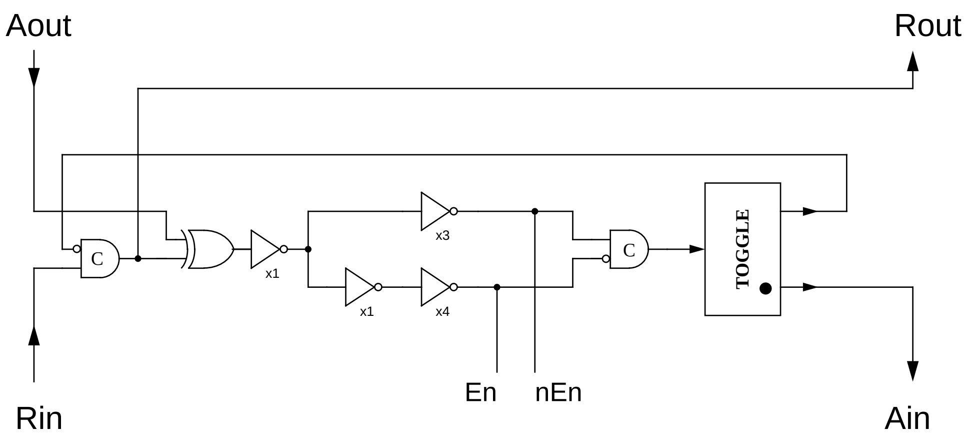


Figure 3-26: Fast-forward Micropipeline control

Another constraint on the use of this type of control structure is that the acknowledge out (*Aout*) must not arrive back before the stage has finished closing and issued the appropriate *Ain* signal. It is usually quite simple to ensure this in the environment. For example, connecting this to another micropipeline stage should ensure that there is sufficient safety margin on the acknowledge back signal (*Aout*).

3.4.5 Power considerations

For small data bundles the power consumed is dominated by the complexity of the control logic. The **Capture-Pass** with the simpler control will therefore be more efficient. However, the power consumed by large data bundle stages is dominated by the energy required to switch the heavily loaded data latch enable lines.

In the **T-Latch** based design the enable lines are switched twice per cycle, once for closing the data latches and one again for opening them. In the **Capture-Pass** design, the *Capture* lines are switched once to capture the data and the *Pass* lines are also switched only once to enable the latches to become transparent again. For the **Capture-Pass** this gives a total of two transitions on heavily loaded lines. Both design styles therefore switch a heavily loaded line (and its complement) twice per cycle.

By examining the load per data bit, it can be noted that the **T-Latch** has one transmission gate load whereas the **Capture-Pass** has two transmission gate loads per bit (for both *Capture* and *Pass* control wires). If the transmission gates in both styles are a similar size then the capacitive load on the *Capture/Pass* control wires will be twice that of the **T-Latch** enable wires.

The energy (*E*) required to switch a node is given by $E = \frac{1}{2}CV^2$ where *C* is the capacitance of the node and *V* is the voltage swing the node is switched through [Mead80, chapter 9]. As the **Capture-Pass** capacitive load is twice that of the **T-Latch**, the energy required to switch the former is twice that of the latter. This indicates that the **T-Latch** is more power efficient for larger data bundles.

When closed, the **T-Latch** does not respond to fluctuations on the input, so no internal power is dissipated as a result of these transitions. However, when a **Capture-Pass** is in the “captured” state, the input is connected to the opposite branch, so any changes on the input

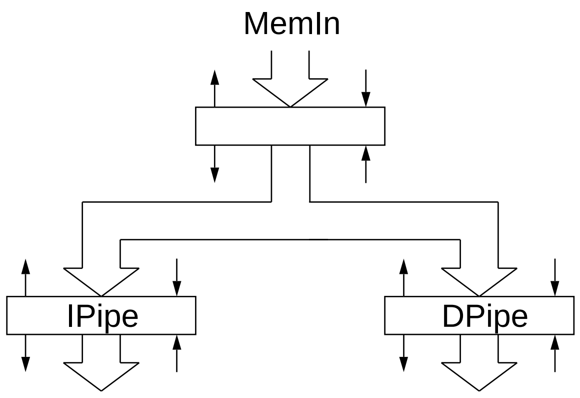


Figure 3-27: Micropipeline fork

cause internal transitions on the branch connected to it. This in turn causes power to be dissipated. Therefore the **T-Latch** is more power efficient, when closed, than a **Capture-Pass.**

Another interesting issue regarding power consumption stems from the fact that Micropipelines are defined to be transparent when empty. By examining a pipeline that has a fork, such as the one shown in figure 3-27, a technique for reducing the power consumption can be demonstrated. Here when both branches are empty they are transparent, so any transitions on the input (*MemIn*) will propagate through both branches of the fork and on towards any processing logic. Although the logic may not be activated for evaluation purposes any transitions cause power to be dissipated in both branches. If most of the incoming data (*MemIn*) is intended for one of the forks, for example the instruction pipeline (*IPipe*), then all instructions destined for the *IPipe* will also cause transitions in the data pipeline (*DPipe*) thus wasting power. If the top stage of the *DPipe* is constructed such that it is opaque when not in use, this would prevent any unwanted transitions dissipating power in the *DPipe*. When data has to enter the *DPipe*, the blocking stage is opened to allow the data in and then closed again to latch the data safely in the top of the pipeline.

It is possible to reconfigure the **T-Latch** Micropipeline control so that it is a normally blocking stage. This is shown in figure 3-28. The incoming request (*Rin*) propagates through the **C-Gate** and **XOR** in a similar fashion to the standard **T-Latch.** The latches are then forced open to allow the data in. The opening is detected by the second **C-Gate** and the **TOGGLE**. The *Dot* output of the **TOGGLE** is fed back to the **XOR** to close the latches again. Once the latches are closed this is again detected and the event from the *Blank* output of the **TOGGLE** is used as the *Rout* and *Ain*. The next *Rin* can be processed as soon as an *Aout* is received from the successor stage.

The penalty for using a blocking latch control is that the forward latency is approximately double a standard **T-Latch** design (the blocking latch must be first opened and then closed). The forward latency can be improved by connecting *Rout* to the *Dot* output of the **TOGGLE** (as the *Dot* output indicates the latch is open). This type of optimization can be justified by similar arguments and constraints as used for the fast-forward latch control.

3.4.6 Choosing an implementation

The most effective style and configuration of Micropipeline implementation is dependent on the intended application. AMULET1 contains a large number of 32-bit Micropipeline

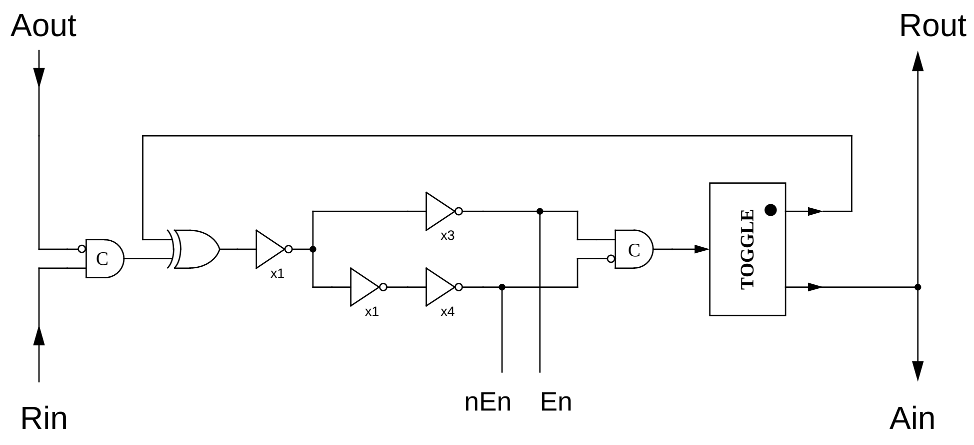


Figure 3-28: Blocking Micropipeline control

stages, so it was felt that the area penalty of using a **Capture-Pass** would be too great. A retrospective investigation [Pave92b] revealed that the data processing part would have been 25% bigger had **Capture-Pass** latches been used.

The exact consequences of using the lower performance **T-Latch** based design is not yet known. It is likely that this may prove to be one of the performance limiting factors of the whole design. It is for this reason further work is being undertaken to investigate whether it is possible to combine the area efficiency of the **T-Latch** design with the simpler, high performance control of the **Capture-Pass** design style.

Chapter 4 : The asynchronous ARM

To investigate the suitability of the Micropipeline methodology for the design of complex systems, an asynchronous implementation of the ARM processor was developed. The asynchronous ARM (AMULET1) is object code compatible with the 32-bit ARM6 and addresses several difficult areas.

4.0.1 The ARM processor

The ARM processor employs a simple and efficient RISC architecture. It was originally designed at Acorn Computers, Cambridge, England in 1983-84. Its main features are:

- A small and simple design with fewer than 35,000 transistors.
- A RISC load/store architecture, with support for efficient block data transfers.
- All instructions can be conditionally executed.
- Low cost because of its small size.
- Very low power, again because of its simplicity and small size. (One of the marketing metrics used by ARM Ltd. is that it delivers 100 MIPS/Watt).

The motivation behind the original design and a detailed description of the architecture are described elsewhere [Furb89, VLSI90], a brief overview of the processor is presented in appendix B.

4.0.2 Implementation challenges

The asynchronous implementation was required to be code compatible with the 32-bit ARM6 and therefore had to address the following difficult areas:

1. Interrupts - The ARM supports two levels of interrupts known as IRQ (Interrupt ReQuest) and FIQ (Fast Interrupt reQuest). Both use level sensitive input signals; FIQ has higher priority.
2. Exact exceptions - The ARM processor supports virtual memory (VM) systems and is therefore able to handle data aborts, for example, caused by a page fault in a VM system.
3. Block data transfer - The ARM can load or save multiple registers from the current working set with one instruction. This requires support for multi-cycle instructions.
4. Conditional instructions - Every ARM instruction is conditionally executed depending on the state of the processor status flags.

The above list presents some of the more challenging features, each of which on its own is non-trivial; handling all of them together can become very complex. For example, a data abort can occur in the middle of a load multiple operation where half the registers have already been reloaded before the exception is raised. The load multiple instruction must be re-started after the cause of the exception has been removed. This is particularly difficult when the base register (which specified the memory address used in the transfer) has itself already been overwritten (e.g. in a context restore) before the exception was flagged.

4.0.3 Differences from the ARM6

The asynchronous organization of AMULET1 differs from the ARM6 in the depth of pipelining employed. The ARM6 has a three stage pipeline (see appendix B): fetch, decode and execute, whereas AMULET1 employs a much greater depth of pipelining. In particular, the execute phase of AMULET1 is sub-partitioned into a further three pipeline stages (register read, shift/multiply and ALU). The exact detail of the pipeline structure is discussed later.

A small number of architectural features were not implemented or do not conform exactly to the ARM6 specification due to the limited resources and time available. The features not implemented include:

- Co-processor instructions.
- The ARM6 26-bit compatibility mode which allows the ARM6 to behave as if it was a 26-bit ARM2.
- The multiply-with-accumulate instruction *MLA* (although multiply without accumulate is implemented).

All the co-processor instructions take the undefined instruction trap and their functionality is emulated in software by the trap handler (the first synchronous ARM also used this approach).

There are several obscure corners of the instruction set where the asynchronous implementation deviates from the specification because the particular “features” were defined as an artifact of the synchronous implementation and have little or no practical use.

4.0.4 Processor interface

The asynchronous implementation of the ARM employs an asynchronous Micropipeline interface to the environment. Figure 4-1 shows how a system containing an asynchronous ARM is configured. It is assumed that the input/output forms part of the memory system. The memory management unit (MMU) and the memory have not been implemented as VLSI circuits at this stage and are beyond the scope of this thesis. The asynchronous interface of AMULET1 has the following signals:-

- The output bundle which contains a memory address, control information and write data, if a write operation is being performed.
- The input bundle which returns read data requested from memory back to the processor.
- The interrupt requests, memory abort response and processor reset.

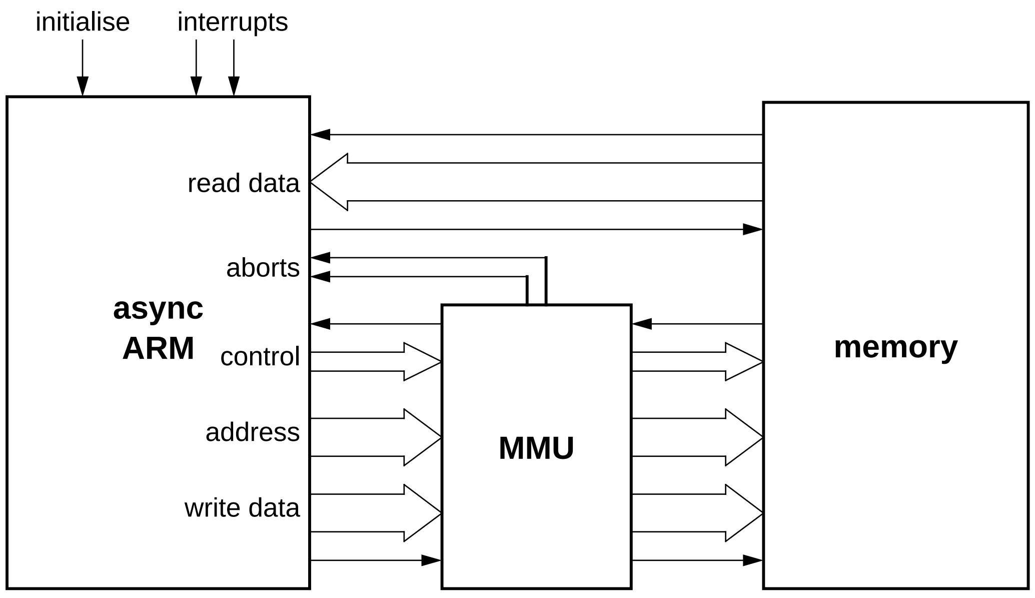


Figure 4-1: The processor interface

The control information in the output bundle consists of a number of bits which include read enable, write enable, a bit to indicate whether an instruction or data is being fetched (opcode flag), a privilege mode bit so that memory protection can be implemented and 2 bits indicating whether sequential address behaviour is likely (and hence whether the memory access can use the faster page mode of DRAM and some cache memories).

The processor design makes no assumption about the memory hierarchy employed. The only constraint that must be enforced is the sequential ordering of memory accesses so that data returns to the processor in the requested order. The time taken for a memory access is not critical for correct operation, if the Micropipeline protocol is obeyed. For example, a system with a cache memory would exhibit a fast response for a cache hit and a much slower response for a miss. However, cache hits must not be allowed to overtake cache misses or the sequential ordering constraint would be violated.

The ARM6 architecture requires support for virtual memory. The method used to handle page faults is that of an exact exception. For every data access, the state of the processor is preserved until it is known that the access will be successful. For each data access, the MMU produces a fault/no fault response that signals back to the processor either to cause instruction processing to resume (if there was no fault), or to enter the exception handling software, (if there was a fault). The response time of the MMU determines how soon the processor can resume executing instructions, so a fast response time is required for optimum processor performance. The response from the MMU is signalled back to the processor in a dual-rail encoded format.

Aborted instruction-prefetch accesses need not signal back to the processor in the same way, as they do not affect the current state of the processor. The non-valid instruction data can be tagged as invalid when read into the processor. The exception handler is then called when the instruction reaches the primary decode.

AMULET1 also supports the two level-sensitive interrupts of the ARM6 so that it can be used with conventional peripheral chips. The level sensitive model of interrupts gives the processor no information regarding the time taken for the peripheral to release the interrupt line, so care must be taken in the exception handling routines to ensure that the same interrupt does not cause multiple exceptions.

The final input to AMULET1 is the initialisation pin. This is used to reset the state of the processor; its release causes the processor to begin issuing sequential instruction addresses to the memory starting from address zero.

4.1 Processor organization

The internal organization of AMULET1 is shown in figure 4-2. This shows the four main areas of the processor and how they are interconnected. A brief description of each of the areas is presented in this section; more detail is provided later in the thesis.

4.1.1 Address interface

The main function of the address interface is to issue instruction prefetch requests to maintain a steady flow of instructions into the processor. The address interface can autonomously issue sequential instruction addresses with the aid of an internal incrementer.

Data transfer operations use the address interface to generate the data transfer address and the multiple data transfer instructions also use the incrementer in the address interface to generate the sequential data addresses required.

The address interface supplies a PC value to the instruction currently being executed to be used as an operand (via the *PC Pipe*). This is needed because the program counter forms the ‘general purpose’ register *R15* in the ARM architecture. The address interface is described in more detail in chapter 6.

4.1.2 The register bank

The register bank contains 30 general purpose registers, sixteen of which are accessible at any one time. It also provides storage for the 5 Saved Processor Status Registers (SPSRs) as specified in the ARM architecture definition. To access the contents of the register bank, two read ports are connected to the *A* and *B* operand buses; a single write port is provided for modifying the contents. Internally the register bank has mechanisms for supporting multiple outstanding write operations and managing inter-instruction dependencies. The detail of these mechanisms is described in chapter 5.

4.1.3 The execution unit

The execution unit is the computational core of the processor. The data read from the register bank can be multiplied using an autonomous 3-bits-at-a-time carry-save-adder multiplier. If multiplication is not required then this logic is simply bypassed. The ARM architecture specifies that one of the register operands can optionally be shifted for certain classes of instruction. To support this, a barrel shifter is attached to the *B* operand bus and its output fed to the ALU. The ALU performs all other logic functions and contains a simple ripple-carry adder with data dependent completion signalling. The output of the ALU can be transferred onto the write bus to route results back to the register bank. The write bus is

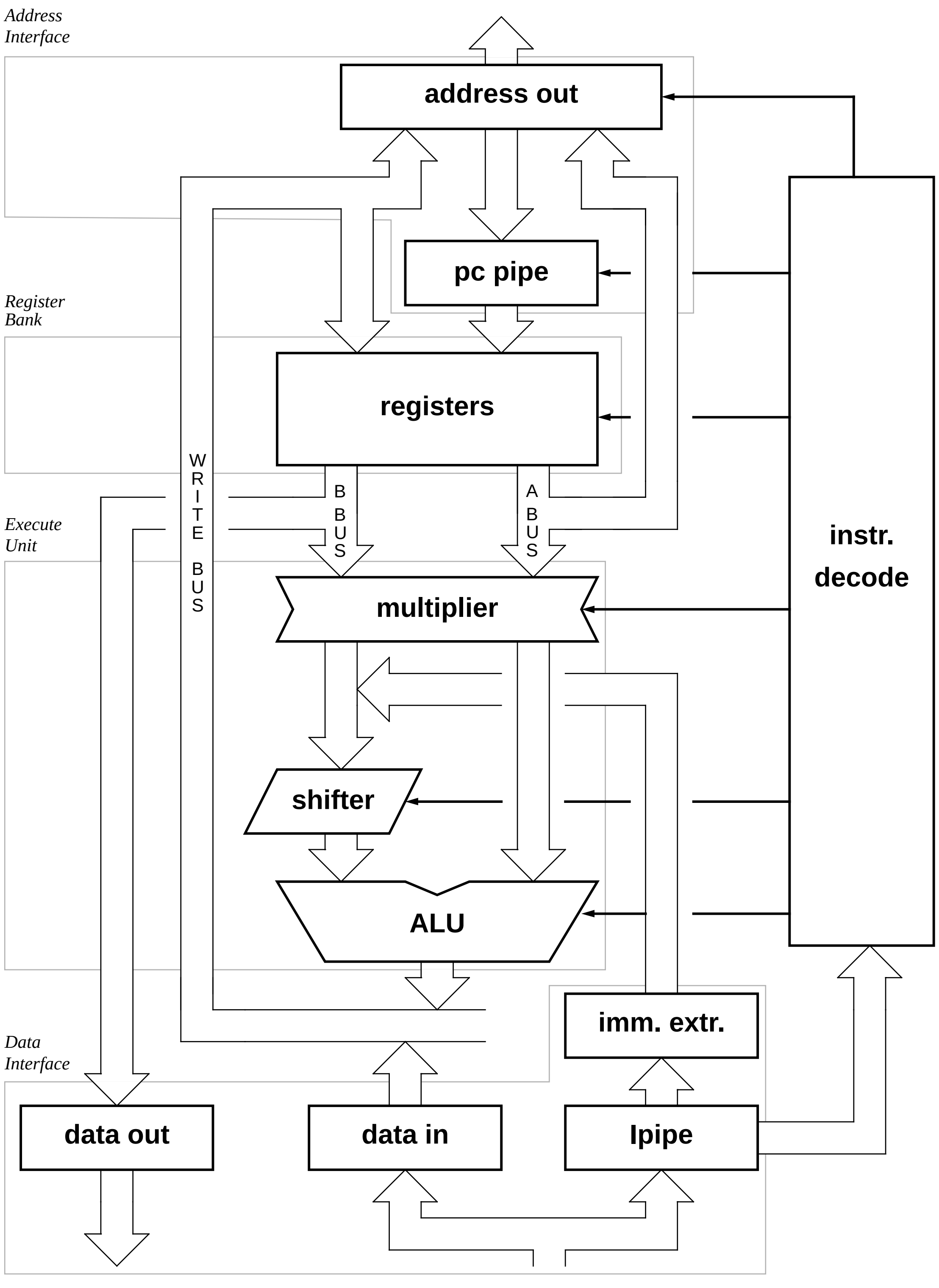


Figure 4-2: Processor organization

shared with the data interface which uses it for incoming data from memory; control of the bus thus requires arbitration (data arrives asynchronously to the ALU result). The execution unit is described in more detail in chapter 7.

4.1.4 The data interface

The data interface manages the flow of data between the processor and the memory subsystem. For write operations it provides a FIFO for storing data waiting to be written to memory (*data out*) and has the capability to replicate the least significant byte into all byte positions across the word (for byte write operations).

Incoming values can be either new instructions for decoding or data destined for the register bank. Instructions are stored in a FIFO prior to execution (*Ipipe*). If the instruction specifies that an immediate value is required then a copy of the instruction is passed to dedicated hardware (*imm. ext.*) which extracts the immediate value from the instruction word.

Data destined for the register bank passes through the *data in* section. This can rotate the incoming data word by byte quantities. Data loads from non-word aligned addresses are performed by loading from the word boundary and then rotating the data word until the addressed byte is in the least significant byte position. *Data in* also contains logic to enable the processor to operate in “little-endian” or “big-endian” modes.

4.2 Pipeline organization and control

Figure 4-3 shows how the processor is divided into pipeline stages; each grey box represents a pipeline latch. The execution phase is partitioned into three stages: register bank, shift/multiply and the ALU. Each data processing stage has a corresponding decode and control stage.

During instruction execution primary decode performs the complete decoding for the register bank and the initial decoding for execution stages 2 and 3. Once the primary decode is complete, the control information is latched along with the corresponding value of the PC. While the operands are being read out of the register bank the secondary decoders for stages 2 and 3 decode the instruction further.

Although the control and data latches are drawn in alignment, the control and datapath only synchronize when they reach their destination stage. Decode 2 only synchronizes with the data when it reaches the shifter/multiplier and decode 3 only synchronizes when it reaches the ALU.

The extra pipeline latch after the ALU is provided to allow the Current Processor Status Register (CPSR) to be accessed; it also decouples the ALU from the result writeback phase.

4.2.1 Dynamic pipeline structure

To reduce their sizes and transistor counts, the register bank, multiplier, shifter and ALU all use dynamic structures. Each stage employs dynamic logic with an output latch for storing the result. The output latch of one stage is the input latch of the next (see figure 4-4(a)). This is similar to the canonical Micropipeline with processing shown in figure 3-6, except that the dynamic logic imposes an additional constraint: the output latch must be empty before

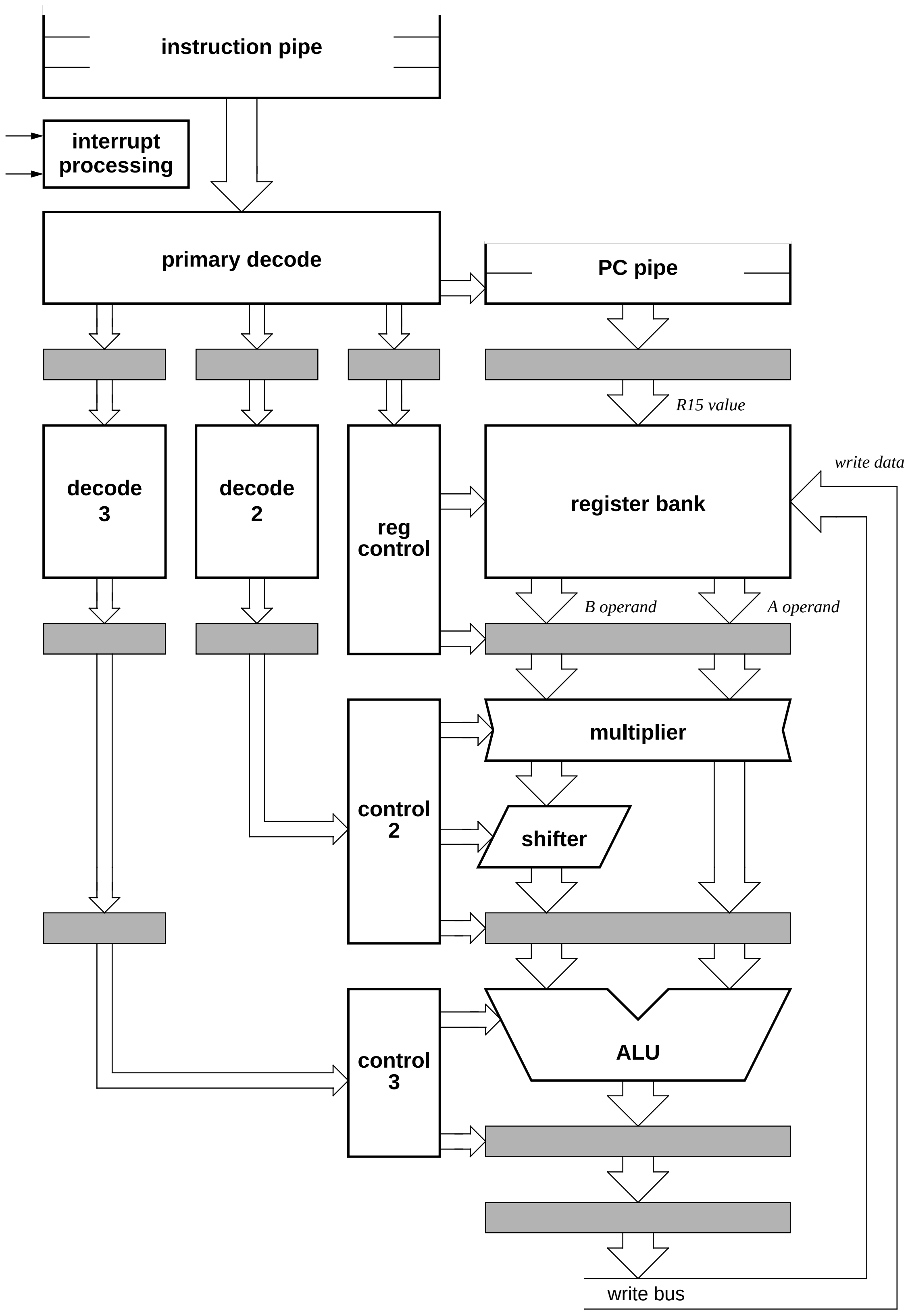


Figure 4-3: Pipeline organization and control

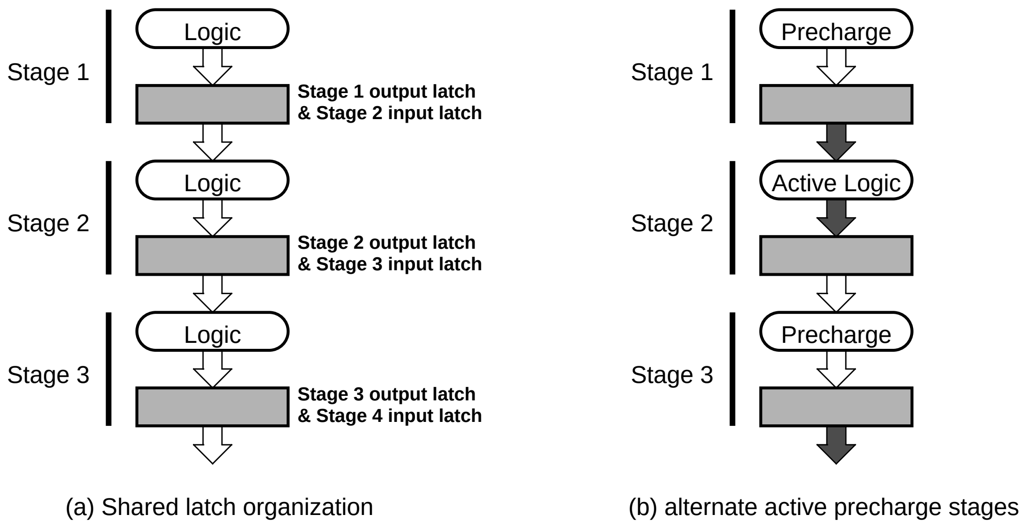

Stage 1
Logic
Stage 1 output latch & Stage 2 input latch
Stage 2
Logic
Stage 2 output latch & Stage 3 input latch
Stage 3
Logic
Stage 3 output latch & Stage 4 input latch
(a) Shared latch organization
Stage 1
Precharge
Stage 2
Active Logic
Stage 3
Precharge
(b) alternate active precharge stages

Figure 4-4: Dynamic pipeline structure

the evaluation of the logic begins to ensure the result can be latched before leakage renders it invalid.

When any stage is active it needs both to be able to write into the output latch and also to ensure that the input does not change until the computation is complete, therefore both latches are involved in the activation of the stage. At any instant this means that only alternate stages may be active. (Note, if no results are removed from the bottom stage, the pipeline will backlog with each stage storing its result value in its output latch; in this state the pipeline is fully occupied)

Having only half the pipeline stages active at any one time seems to incur a performance loss; however, the dynamic stages require a precharge phase, so while one stage is active the adjacent stages can precharge (figure 4-4(b)). If an extra latch were provided to decouple the two stages, the pipeline could be fully active because the input and output latches need no longer be shared. However, data flow from one stage to the next would still have to wait for the next stage to precharge. This, along with the extra pipeline latch, would increase the latency through the pipeline.

The dynamic shared latch pipeline with only alternate stages active can be used to advantage in the ALU and shifter. The shifter needs the current carry flag (for rotate with carry) which can be changed by the ALU, but because the shifter and ALU are adjacent stages only one of them can be active at any time. While the ALU is executing and possibly calculating the new flag values, the shifter is in precharge. When the ALU is complete and the flags are valid, the ALU input latch is released and the ALU enters precharge. This allows the shifter to begin execution. The flags are now valid and cannot be changed by the ALU, so they can be used directly by the shifter without the need for complex synchronization between the ALU and the shifter.

4.3 Instruction mapping

The mapping of the ARM instruction set onto the asynchronous organization can be described by examining the datapath activity of each class of instructions.

4.3.1 Data operation

The simplest class of ARM instruction is the data processing operations. These can take two operands, perform one of sixteen operations and then write the result to a register.

Figure 4-5(a) shows the datapath activity for a data operation that uses two register operands. Here the two operands are read from the register bank onto the *A* and *B* operand buses (see figure 4-2 for bus naming convention). The multiplier is configured so that the operands bypass the multiplier logic without activating it. The ARM architecture specifies that one of the operands can be shifted by an arbitrary amount so the *B* bus is passed through a barrel shifter to perform this operation (the type of shift performed is specified in the instruction). The two operands then enter the ALU where they are combined according to the data operation instruction being executed. The ALU then takes control of the write bus and sends the result back to the register bank (unless the destination is the program counter[1] - *R15*).

If the destination is *R15*, the result is steered to the address interface instead of the register bank and the operation is similar to that of a branch (described in the next section).

If one of the operands is *R15* then the bottom entry from the *PC pipe* is multiplexed onto the required operand buses. The correct value of the PC is supplied before instruction execution begins, therefore no lock mechanism is needed to prevent register hazards (see section

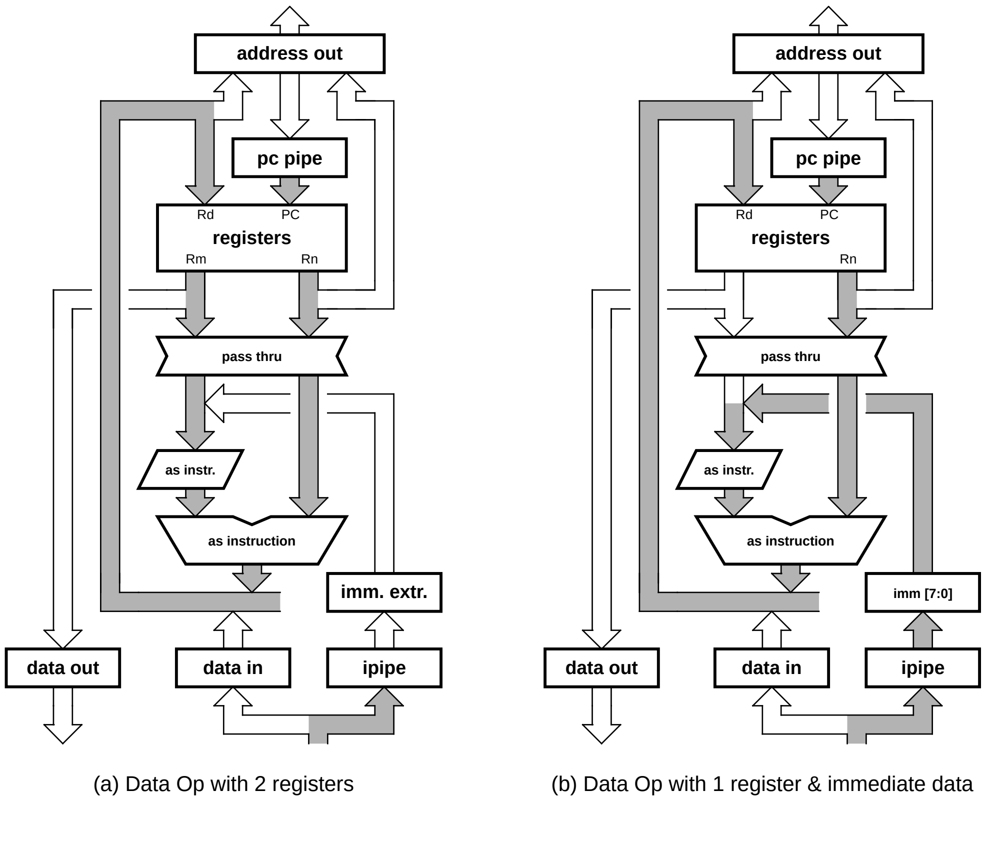


Figure 4-5: Data operation datapath activity

1. The program counter can be accessed as a general purpose register (R15) in the ARM architecture.

5.1.2). Any change of instruction flow (and hence PC availability) is managed at a higher level of control.

When only one register is required, the other operand defaults to the PC because this will not stall waiting for data to return to the register bank. The PC value is then discarded. It would have been more power efficient to read just one register, but this would have required extra bypass logic to implement whereas the solution adopted requires no additional circuitry.

When one of the operands is an immediate value, as shown in figure 4-5(b), then only one register is read from the register bank. The other operand is an 8-bit unsigned value extracted from the instruction word and then multiplexed onto the input of the shifter. Apart from the source of the operands, the overall operation is as described previously.

4.3.2 Branch operation

In the ARM architecture, the branch (*B*) is a PC relative jump. The instruction word contains a 24-bit immediate offset which is word aligned, sign extended and added to the current PC value to achieve the branch target address. The ARM also specifies a subroutine call instruction, branch and link (*BL*), which is similar to the branch but the address of the next instruction is saved in register *R14* to facilitate the subroutine return.

Figure 4-6(a) shows the branch operation and the first cycle of a branch and link. Here the PC value is read from the register bank onto the A operand bus. The offset is extracted from the instruction word and sign extended to 32-bits. This value is then fed into the shifter

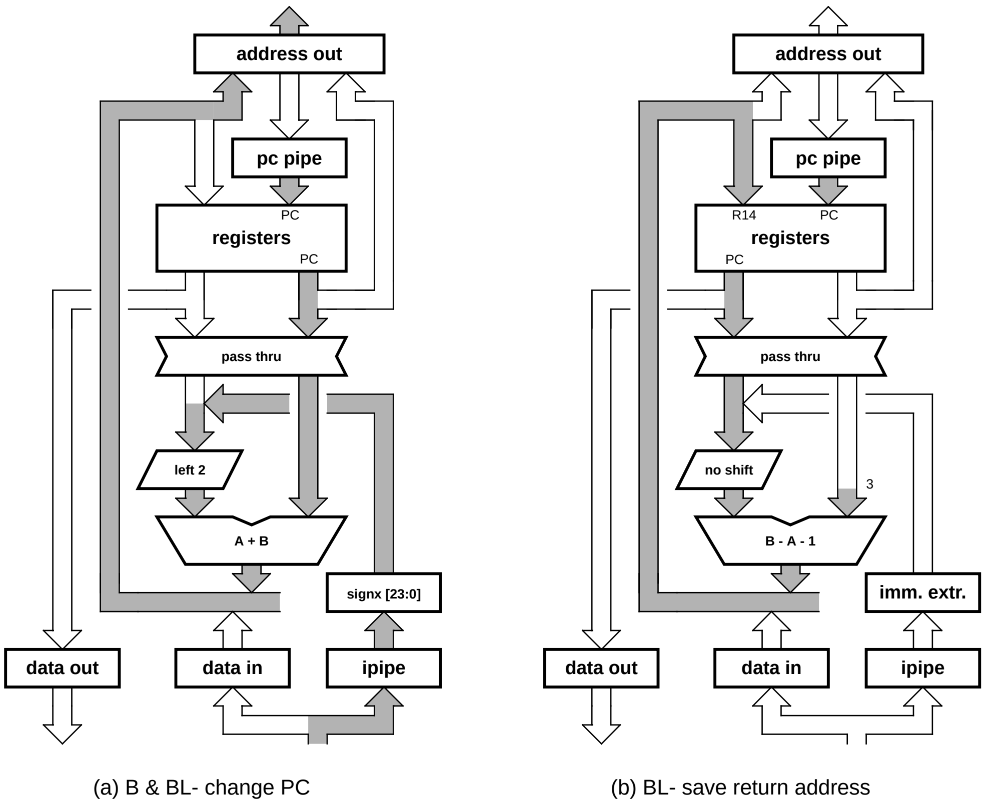


Figure 4-6: Branch & branch and link datapath activity

where a left shift of 2 is applied to word-align the value before it is added to the current PC value in the ALU. The ALU takes control of the write bus and passes the target address to the address interface. Here the target address enters into the prefetch loop and replaces the original prefetch address (see chapter 6 for further details). Prefetching (not shown on the diagrams) then resumes from the branch target. As a number of instructions may already have been prefetched, these must be thrown away before instructions from the new stream arrive. This is achieved by the instruction *colour* mechanism described in section 4.4.3.

A branch operation is concluded after the above cycle; branch with link however requires an extra cycle to save the return address (i.e. the address of the next instruction). The PC value delivered when *R15* is accessed is PC+8 (see section 4.4.1). The address of the next instruction is PC+4, so the value of *R15* read from the register bank is decremented by 4 before it is written into the link register (*R14*), figure 4-6(b) shows the detailed operation of this. The *R15* value is read from the register bank on the *B* bus and passed through the multiplier and shifter unchanged. Four is subtracted by forcing the bottom two bits of the *A* operand bus High (three in binary) and then performing a subtract with borrow, hence achieving B-A-1 or (PC+8)-3-1= PC+4. The PC+4 value is written back to the register bank via the write bus in a similar fashion to data processing operations.

4.3.3 Multiply operation

The multiplier in AMULET1 uses a shift-and-add technique with carry-save adders (see section 7.1). The multiplier accepts two input operands from the register bank and internally performs the multiplication of the two. When the operation is complete the partial sum and partial carry are placed on the two output buses. These are then added together in the ALU to form the result. This result is written back to the destination register. The datapath activity is shown in figure 4-7.

4.3.4 Load data operation

The data transfer instructions in the ARM allow the programmer to specify a variety of auto-indexing options including pre-/post-index and increment/decrement of an offset from the base address. The operation can be split into two phases; the address calculation and the data transfer. The address calculation will be described first.

Figure 4-8(a) shows a post-index load with a register offset and writeback (in fact, post-index is defined always to write back). Here, the address from which the data is loaded is the value contained in the base register before it is modified. To perform the load, the base register (*Rn*) is sent via a special route directly to the address interface where it interrupts the instruction prefetching to issue the data load address, at the same time the base register is forwarded to the ALU, ready for the base update calculation. The offset register read onto the *B* bus can optionally be shifted before being added to or subtracted from the base in the ALU. The load cycle completes with the modified base value being written back to the register bank.

Figure 4-8(b) shows a pre-index load with register offset and writeback specified. The pre-index designation indicates that the base is modified before being used as the data load address, so the writeback value is the same as the load address. The operation begins with the reading of base and offset from the register bank which are forwarded to the ALU after

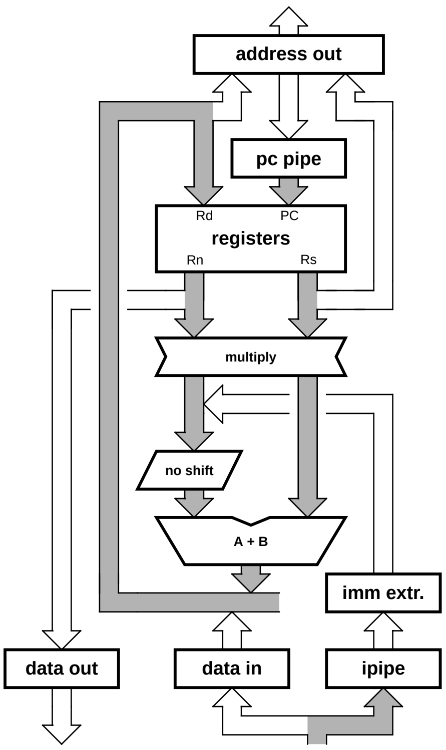


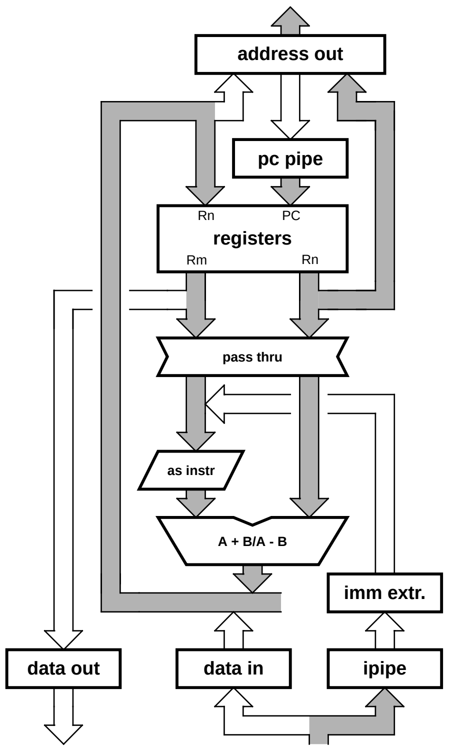
Figure 4-7: Multiply datapath activity

optional shifting of the offset. The base modification calculation is performed and the result transferred to the register bank (if writeback is specified) and to the address interface to be used as the load address.

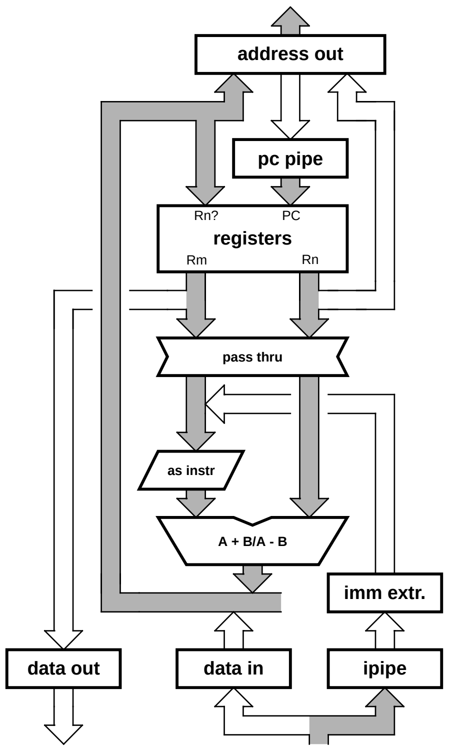
The ARM architecture specifies that an immediate value can be used as an offset. The datapath activity of the immediate value (not shown) is similar to that shown in data operations except that the offset this time is 12 bits rather than 8 and no shift can be applied to the immediate value.

In the ARM, non-word aligned byte loads are implemented by retrieving the data word from the word boundary and then rotating the value obtained so that the addressed byte is in the least significant byte position of the word. In the ARM6, the rotation is performed in the barrel shifter as the data passes through on its way to the register bank. In AMULET1 instructions which do not depend on the loaded data can continue to be processed while the data is being loaded (see section 5.4.1). This means that the barrel shifter could be busy and unavailable to rotate the incoming data, hardware is therefore provided in the data interface to perform the required byte rotation.

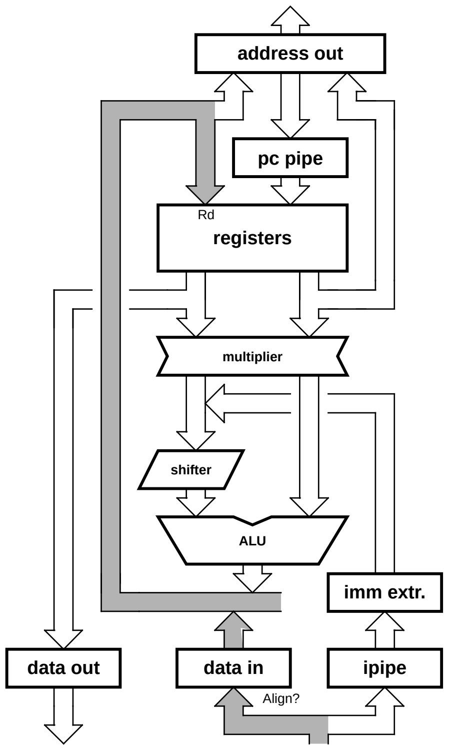
The arrival of data is completely asynchronous to the internal operation of the processor, so when *data in* wishes to use the write bus to transfer data to the register bank, it must arbitrate with the ALU for control of the bus. If data is destined for any register, other than *R15*, it is routed to the register bank (as shown in Figure 4-8(c)). If the destination is *R15* then the loaded data is passed straight to the address interface to be used as the new PC value (as shown in Figure 4-8(d)).



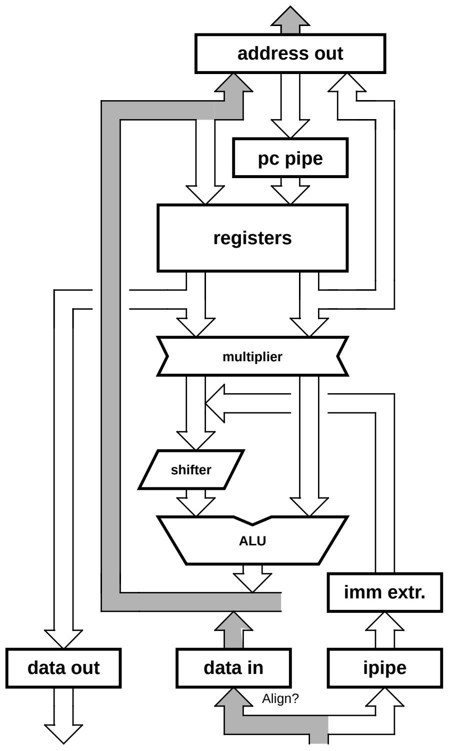
(a) Post-index LDR, register offset & writeback



(b) Pre-index LDR, register offset & writeback



(c) LDR data writeback, dest ≠ R15



(d) LDR data writeback, dest =R15

Figure 4-8: Load register datapath activity

4.3.5 Store data operation

The store data operation has the same auto index options as the load instruction class. The single cycle immediate offset store will be described first and then the more complex two cycle register offset case will be analysed.

With an immediate offset, only 1 register (*Rn*) is needed from the register bank for the base address calculation. This allows the other register bank read port to be used to transfer the store data to the data interface (*data out*). The calculation of the address for store with immediate offset is similar to the address calculation described for load operation. Post index store (figure 4-9(a)) uses the direct connection from the register bank to the address interface and pre-index store (figure 4-9(b)) writes the same data to the register bank and the address interface. If store byte is specified, then the least significant byte of the word is replicated in *data out* so it appears at each byte position. The store data then synchronizes with the store address before being despatched to memory for the operation to be completed.

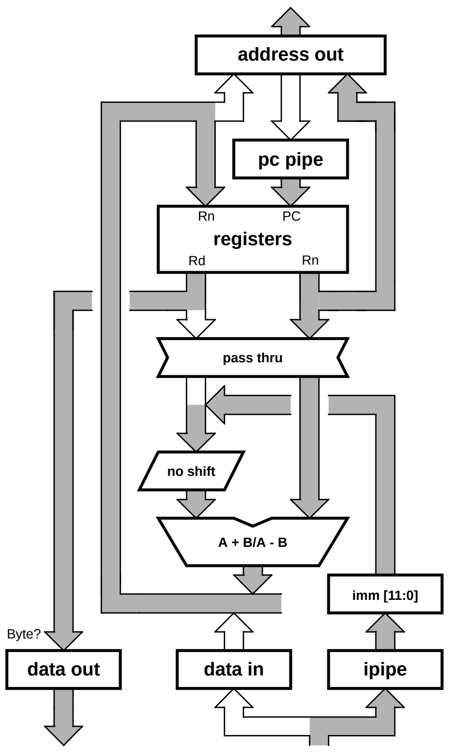
A store data instruction with register offset has a fundamental problem which prevents single cycle operation; three register operands are required but the register bank only has 2 read ports. The three register operands are base, offset and store data. An extra read port could have been added to the register file to make single cycle operation possible but as the store with register offset operation accounts for only 1% of all instructions [Furb89, page 278] the improvement in overall performance would be small. The solution adopted is to perform the base calculation in the first cycle (figure 4-9(c)) and access the data to be stored in the second cycle (figure 4-9(d)).

The cost of the second cycle is hidden to some extent by the depth of pipelining in the execution unit. The store data can be read out of the register bank at the same time as the address calculation is being performed in the ALU. The data route is a simple FIFO (with optional byte replication) so the data propagates quickly and should therefore be available by the time the address is ready to be sent to memory.

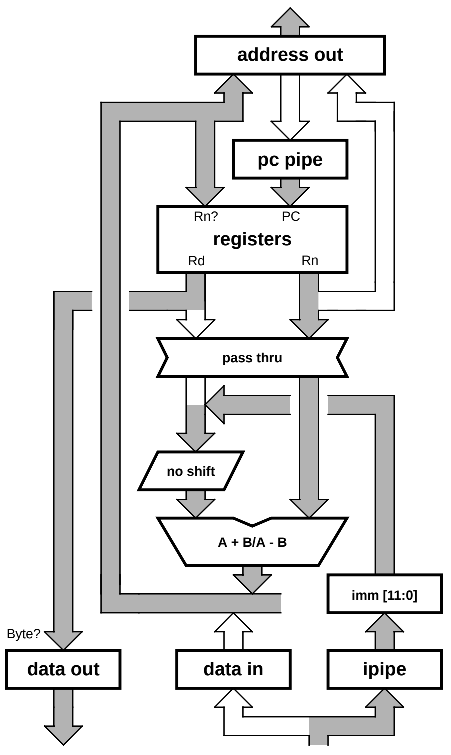
4.3.6 Block transfer operation

The block transfer instructions are by far the most complex instructions to implement. The instructions allow any subset of the sixteen available registers to be loaded or stored and the base register to be auto-indexed. The instruction has options allowing different modifications to the base thus supporting various stack paradigms (e.g. empty/full, ascending/descending stacks). The detailed operation of these options is complex and is described elsewhere [ARM91b].

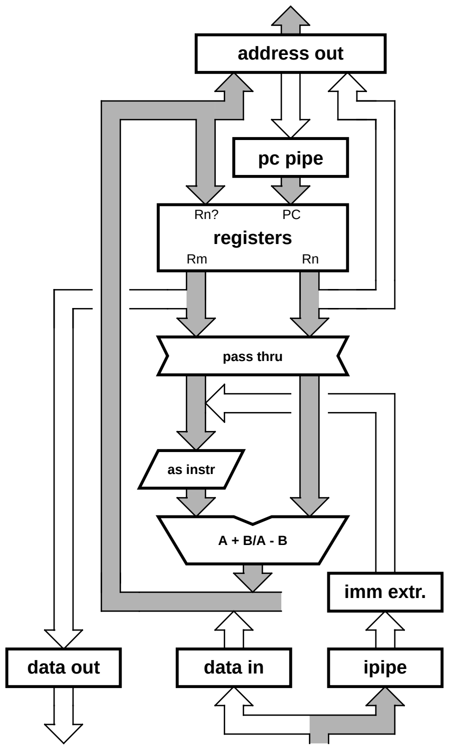
The first cycle of a block transfer instruction calculates the transfer start address. The calculation is performed by reading the base address from the register bank and adding an offset in the ALU (as shown in figure 4-10(a)). The offset is introduced onto the *A* operand bus using a similar mechanism to that described for branch operations. The value of the offset depends whether a load or store operation is being performed and the stacking paradigm selected. For example, an STM will have an offset of 3 for a full ascending stack (add), *Cnt* +3 for a full descending stack (subtract), *Cnt* for an empty descending stack, or 0 for a empty ascending stack (where *Cnt* is the number of registers being transferred).



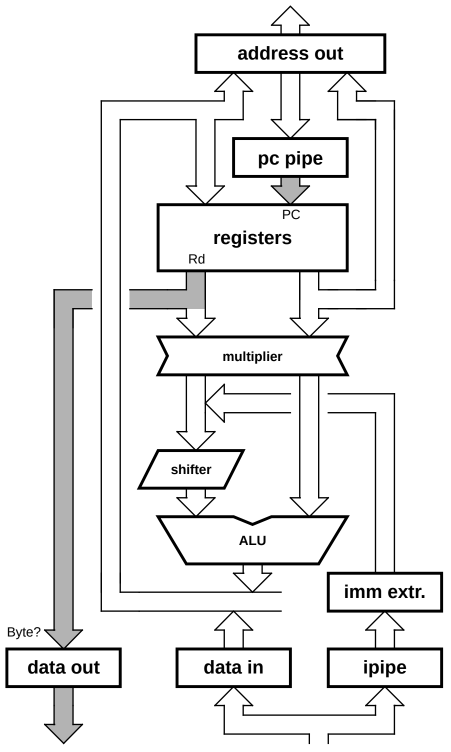
(a) Post-index STR, immediate offset & writeback



(b) Pre-index STR, immediate offset & writeback



(c) Pre-index STR, register offset & writeback,
cycle 1 - calculate address



(d) Pre-index STR, register offset & writeback,
cycle 2 - store value

Figure 4-9: Store register datapath activity

Once the address calculation is complete it is forwarded to the address interface where it waits for synchronisation with the primary decode.

The second cycle of the block transfer calculates the value the base register should have after the instruction has completed (i.e. the modified stack pointer). The cycle takes the base out of the register bank and forwards it to the ALU as in the first cycle. The offset introduced this time on the *A* bus is just *Cnt*, the number of registers to be transferred. This is either added or subtracted depending on whether data is being loaded or stored and whether the stack is ascending or descending. The base is then written back to the register bank. For a load instruction, the base value is held in the output latch of the ALU (*old*) in case it is needed for data abort recovery (described below).

The third cycle of a block transfer stalls until it is known that the instruction is definitely going to be executed i.e. it has passed both its conditional and colour tests. This is done because it is very difficult to cancel the data transfers once they have started (data would have to be retrieved from the *data out* pipeline etc.)

To achieve this interlock, the first cycle signals back to the primary decode to indicate whether the instruction will execute. At the end of the second cycle this signal is checked to determine whether the third cycle can begin or must be discarded.

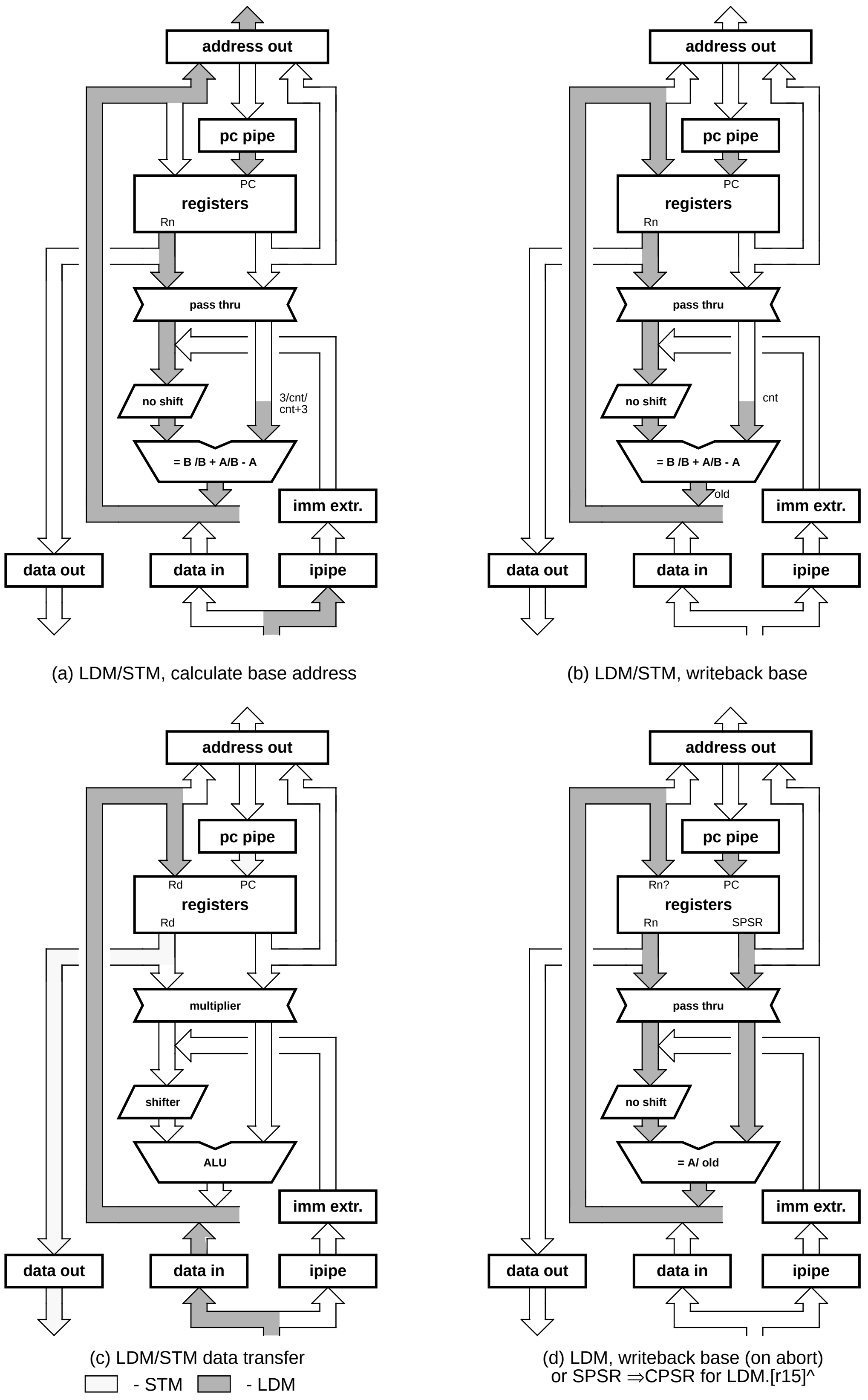
The interlock mechanism could have been placed on the second cycle instead, but this would have caused the execute pipeline to be starved of instructions while the second cycle waited for an acknowledgement from the ALU. By waiting on the third cycle, the second cycle can still proceed and progress down the execute pipe immediately after the first cycle. The cost of the interlock is therefore hidden to some extent because the processor is still executing a part of the instruction. The disadvantage of this approach is that if the instruction does fail its condition test then two cycles have been executed, rather than just one if the more conservative approach had been adopted.

The third phase of a block transfers is the data transfer phase, here the number of cycles is directly related to the number of registers being transferred (one cycle for each register). Figure 4-10(c) shows the data transfer for both load and store multiple instructions. For each register in the data transfer, the primary decode signals to the address interface to generate the memory address of the register being transferred. The next address is generated with the aid of the incrementer in the address interface (the incrementing mechanism is described in more detail in chapter 6). The primary decode generates the address of the register to be transferred.

For store data transfers, the data exits the register bank on the *B* bus and is diverted into the *data out* pipe in the direction of the data interface. The instruction does not activate any further elements on the execution datapath. When the data reaches the data write register it synchronizes with the address generated by the address interface and is dispatched to memory.

For a load instruction, the transfers just involve locking destination registers and signalling to the address interface to generate the load address. The returning data enters the processor in a similar manner to the standard load data instruction.

For both load and store transfers, the signal to the address interface also indicates whether the current transfer is the last. If it is the final transfer then the address is discarded after use as the data address (rather than being incremented ready for the next transfer).



(a) LDM/STM, calculate base address

(b) LDM/STM, writeback base

(c) LDM/STM data transfer

▢ - STM ▨ - LDM

(d) LDM, writeback base (on abort) or SPSR ⇒CPSR for LDM.[r15]^

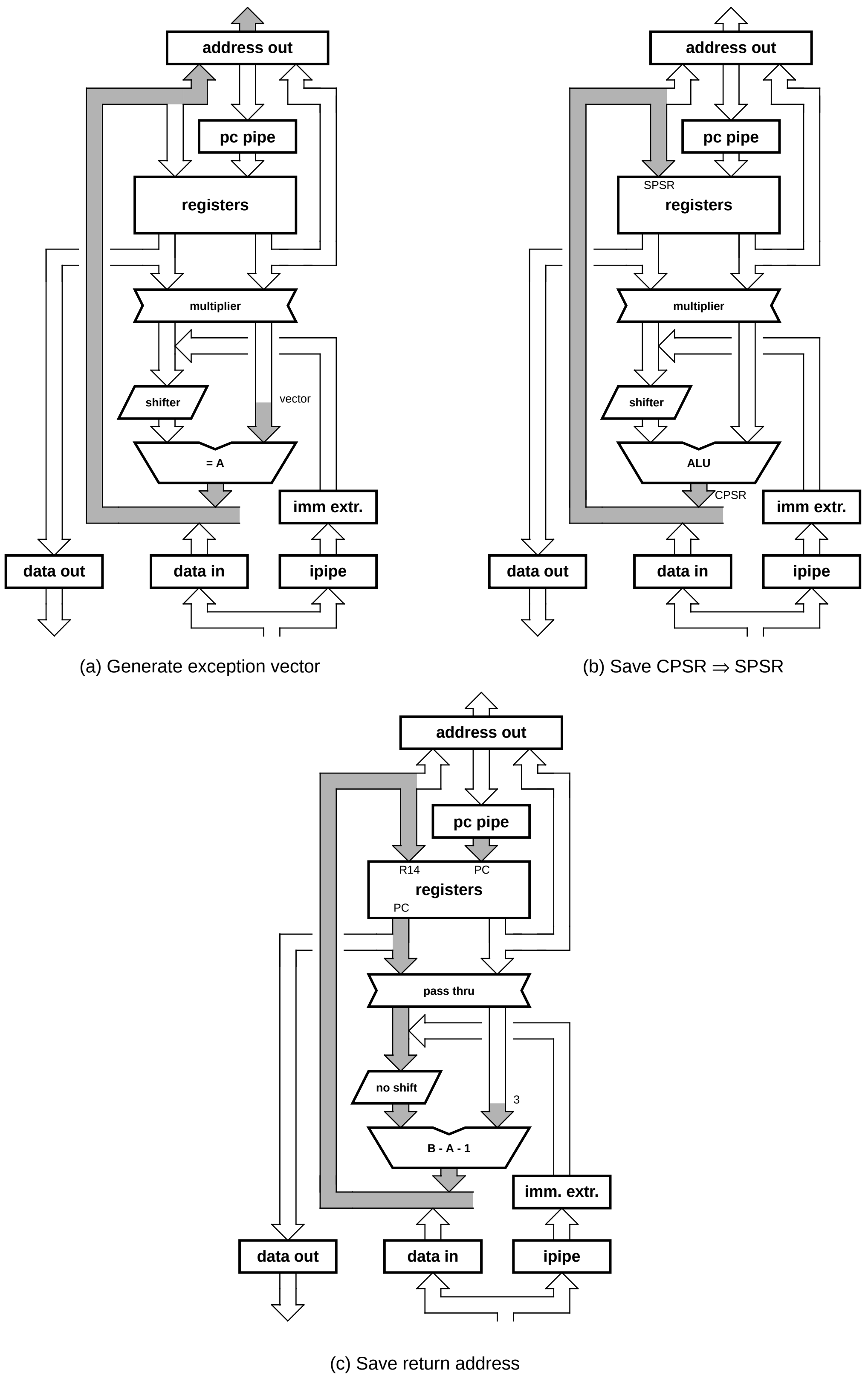
Figure 4-10: Load/Store multiple datapath activity

For load instructions, the signal between primary decode and the address interface also notifies if this transfer is to the program counter (*R15*). This primes the address interface to discard the current prefetch address (after the block transfer is complete) and also forces it to wait for the new program counter, arriving via the *data in* port.

For the store multiple instruction, the transfer is complete after the third phase; however, for load instructions there is another cycle. This final phase (figure 4-10(d)) performs one of two possible functions. The load multiple instruction allows the programmer to specify that *R15* is reloaded with the Current Processor Status Register (CPSR) restored at the same time. The final cycle copies the SPSR to the CPSR in this case. The other possible operation in the final phase is recovery of the base after a data abort. It is possible that the base may have been reloaded before an abort was raised. To restart the instruction, the original base register is needed (although the written back value is sufficient). This is preserved on the output of the ALU in the second cycle (shown as *old*). Therefore, to restore the base, a dummy read from the register bank is sent to the ALU. The ALU ignores its inputs and sends the previously preserved value (*old*) of the base register back to the register bank. It is theoretically possible for this preserved base to arrive back at the register bank before the data for the same register arrives back from memory so the situation can arise that although the base is restored it is still overwritten by the loaded data. To ensure that this cannot happen, the dummy read actually reads the base register, so that if the data destined for the base from memory has not yet returned, it stalls. The read only proceeds when the loaded data is safely in the register, so the base restore can now proceed without fear of being corrupted.

4.3.7 Exception entry operation

Once an exception has been detected by the primary decode, the entry mechanism is the same for all exceptions. The first cycle shown in figure 4-11(a) sends the exception vector address to memory. This is done by performing dummy reads from the register bank and then introducing the exception vector onto the *A* operand bus. This vector is passed straight through the ALU and sent to the address interface to become the new program counter. The second cycle (figure 4-11(b)) preserves the current processor status register (CPSR) by copying it from the ALU into the SPSR of the exception mode the processor is entering. The final cycle (figure 4-11(c)) saves the return address of the instruction that was about to execute before the exception was entered. For everything except data aborts, this is a modified version of the PC value of the instruction extracted from the end of the PC pipeline as for a normal access to *R15*. However, for data aborts, there is a special holding register that preserves the address of the data transfer instruction that aborted (described in more detail in section 6.1). This value is multiplexed onto the output of the *PC pipe* so that it can be used as the return address of the exception handling routine.



(a) Generate exception vector

(b) Save CPSR ⇒ SPSR

(c) Save return address

Figure 4-11: Exception entry datapath activity

4.4 Instruction flow control

There are three main aspects to instruction flow control in AMULET1:

- Instructions must have access to their correct PC values prior to execution in case it is needed as an operand.
- Instructions which fail their condition tests must be discarded.
- Instructions already prefetched when a branch is taken must also be discarded.

All these areas must be addressed to ensure correct operation and to maintain backwards compatibility with the ARM6.

4.4.1 The ARM PC model and the PC pipeline

In synchronous ARM processors, the value of *R15* - when used as an argument - reflects the instruction prefetching which has been done at the time it is used. In a clocked implementation this is well defined and for most instructions yields a value of the current instruction address + 8 (i.e. PC+8). If the PC is used as a *return address*, i.e when it is moved into the link register (during procedure calls) the value written is PC+4. This is the address of the instruction following the current one and is thus the address of the instruction to return to after the procedure is complete. The value PC+4 can be calculated from PC+8 using the ALU on the way to the link register.

In the synchronous ARM, the PC value is taken directly from the prefetch unit. This is convenient but closely couples the architecture to a specific implementation. In the Micropipelined implementation there is no direct coupling between the prefetch unit and the ALU input. It is therefore not possible to determine how much prefetching has occurred, and hence the relationship between the PC value in the prefetch unit and the required value is not fixed.

For AMULET1 to be code compatible with an ARM6, *R15* must have the correct value; to achieve this each instruction has a PC value associated with it. The instruction and its corresponding PC value synchronize just before execution of the instruction begins. This is implemented by keeping a FIFO (first-in, first-out queue) of PC values and matching them with the instructions as they return from memory. This FIFO is called the “PC pipeline”. Instructions returning from memory are stored in the instruction FIFO prior to execution. Figure 4-12 shows a high-level representation of this operation; this is discussed in more detail in section 6.1.4.

4.4.2 Condition code evaluation

All ARM instructions are conditionally executed depending on the setting of the Current Processor Status Register (CPSR) arithmetic flags, (*N*egative/signed less than, *Z*ero, *C*arry/not borrow/rotate extend, o*V*erflow) and the instruction word bits [31:28]. Thus a comparison of the flag settings and instruction bits [31:28] must be carried out to check whether the instruction should be executed or not. Instructions are typically executed with the ‘always’ condition.

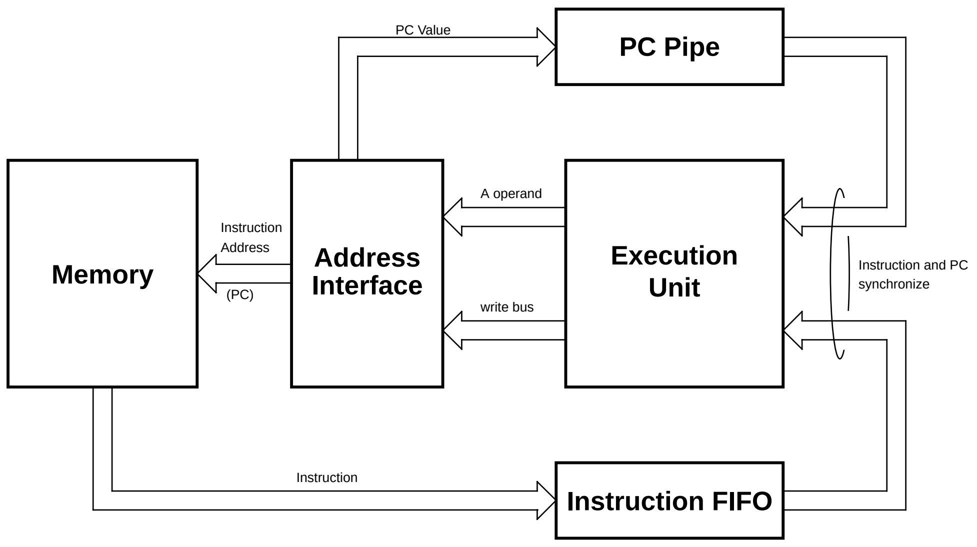


Figure 4-12: Instruction/PC matching

The test of the condition flags is performed in parallel with the second stage of execution (i.e. in parallel with the shift/multiply). A signal is sent from the second stage with the data to the ALU to indicate whether the condition test was successful or not. This is used by the ALU control to invalidate instructions which have failed their test. Instructions that have previously locked a destination register must still return to the register bank to unlock the destination. No data is written back from invalidated instructions; the lock is removed without changing the register contents (see chapter 5 for detail about register locking).

The condition test can be performed in parallel with the shifter because the flags are already valid by this time and cannot change until the ALU is activated. This is a consequence of the dynamic shared latch pipeline structure.

4.4.3 Branch operations

With an autonomous prefetch system, it is not possible to predict how many instructions will have been prefetched when a branch is taken. Prefetched instructions invalidated by the branch must be discarded before instructions from the branch target are processed. This is achieved by maintaining a single bit *colour* flag which changes every time a branch is taken. An instruction has an associated colour bit which indicates the state of the processor colour when the instruction fetch was requested.

When the instruction stream changes, the reference colour changes and hence the colour of instructions subsequently fetched also changes. Each instruction which arrives at the datapath for execution has colour information which indicates whether the instruction was from the original stream (and hence must be discarded) or from the new stream (to be executed). Instructions whose colour differs from the new reference colour are discarded. This continues until instructions from the new stream arrive with the same colour as the reference; these are then executed.

There are two places on the datapath where the instruction colour could be checked: at the ALU or at the primary decode. Each has merits and disadvantages as discussed below.

Colour checking at the ALU

The most obvious position for colour checking is at the ALU, since a mechanism already exists there to discard instructions which fail condition tests; this mechanism could simply be extended to check colour as well. The ALU stage is also the earliest point at which it is known whether an instruction has passed its condition test which determines, in the case of a branch, whether or not the instruction stream will change.

Discarding instructions at the ALU is advantageous if a branch is not taken, because the following instruction will be immediately behind the failed branch in the pipeline. The cost of a branch not taken is the same as an instruction failing its condition test, i.e only 1 cycle.

However, if the branch is taken, instructions to be discarded must still traverse the datapath to have their colours checked at the ALU. The performance of taken branches may be impacted if the time taken to flush the old instruction stream is greater than the branch target reload time. This may be significant if one of the instructions to be discarded is a multiply which can take up to eleven internal cycles in the shift/multiply block (before being discarded at the ALU). However, typically, loading from a non-sequential location is likely to take longer than flushing the datapath.

Another problem with discarding instructions at the ALU is that by the time the instruction reaches the ALU it has already locked its destination registers (if any). Therefore the instruction must still notify the register bank that the destination register should be unlocked, even though its result is to be discarded.

The worst case for both power and performance would be the instruction FIFO being full of prefetched multi-cycle instructions when the instruction stream is changed. Each instruction would enter the datapath, split into multiple cycles, progress down the datapath to the ALU and then be discarded. The activation of the functional units on the way to the ALU would consume power and could take longer than the branch target reload time. Although typical instruction streams may not often exhibit such worst case characteristics, it is still disadvantage of colour checking at the ALU.

Colour checking at the primary decode

The main advantage of checking the colour before any registers are read is that the instruction can then be discarded before entering the datapath. This offers a power saving over the previously described ALU colour checker (no functional units are activated when instructions are being discarded). Another advantage is that the instructions are discarded before entering any computational stages so that even complex instructions can be discarded quickly and efficiently.

As all instructions are conditionally executed, it is not known whether a branch will be taken until its condition codes have been evaluated at the ALU decode. Instructions following the branch cannot be allowed to proceed to the colour checker in the primary decode until confirmation is received whether the branch was taken or not (so that the reference colour can be changed). Confirmation is received when the branch instruction reaches the ALU; the effect of this is that the pipeline stalls at the primary decode waiting for confirmation from the ALU for every possible branch instruction. For taken branches this is very efficient because only instructions which are to be executed from the original instruction stream enter the datapath and hence consume power.

If the branch is not taken (i.e. the branch failed its condition test) then the pipeline between the ALU and the register bank will be emptied for no reason. Investigations into ARM instruction set usage [Jagg89] have revealed that approximately 15-20% of instructions executed are branches, with just over 50% of branches taken. This means that approximately 10% of the instructions executed would stall at the register bank unnecessarily, with the performance penalty that flushing the datapath pipeline carries.

A combined colour checking mechanism

Both schemes described have merit, however each has drawbacks. The solution adopted is a combination of the two. Colour checking is primarily carried out at the ALU decoder. However, when the colour changes, the colour information is transmitted to the primary decode with the aid of an arbiter. Instructions already in progress are discarded at the ALU and instructions waiting in the instruction FIFO are discarded by the primary decode.

Care must be taken to ensure that when instructions are discarded by the primary decode, the corresponding entry in the PC pipeline is also discarded to maintain the correct association between instructions and PC values.

4.5 Exception handling

The ARM architecture specifies four classes of exception that must be handled. These are listed below in order of increasing complexity:

1. Software interrupts and undefined instructions.
2. Instruction prefetch aborts.
3. Hardware interrupts.
4. Data aborts.

For the first class of exceptions, the primary decode simply recognises the bit pattern for the software interrupt (SWI) or undefined instruction and immediately begins the exception entry routine (see section 4.3.7). This is similar to normal instruction execution.

When an instruction prefetch abort occurs the memory responds with a invalid data word marked by an “abort” flag (*Pabt*). This flag is copied into the instruction pipeline along with the invalid instruction where it is queued for entry into the primary decode. When it enters primary decode the prefetch abort flag is detected causing the instruction data to be ignored and the exception entry routine to be entered. The aborted instruction still associates with its PC value, which is read as *R15* by the exception entry routine and stored (after suitable modification) as the exception return address.

4.5.1 Hardware interrupts

The level sensitive hardware interrupts may become active at any time and are completely asynchronous to the processor’s operation. Therefore arbitration is required to prevent any synchronization failure. These interrupts enter the processor between the instruction pipeline and the primary decode (see figure 4-3).

Figure 4-13 shows the arbitration hardware in the event path between the *Ipipe* and the primary decode. Although the interrupts are level sensitive, a change in level which causes the interrupt to become active can be viewed as an event. The transition between levels can be used to trigger a Micropipeline arbiter. This is also a convenient place to manage the reference instruction colour which arrives asynchronously from the ALU when a branch is taken. The robust arbiters described in chapter 3 are used. The operation of the circuit is as follows:

Initially the rising edge of the reset signal (*Cdn*) requests control of all three arbiters in parallel (*r1*). Once all three grants (*g1*) have been received an event is sent to primary decode (*Lreq*) to indicate that the arbiters are locked. Any events arriving on *FIQ, IRQ* or *PCCol* cannot gain control of the arbiters. When the primary decode finishes decoding the first instruction it issues an acknowledge (*Pack*), causing all three arbiters to be temporarily released. If there are any outstanding requests on *FIQ, IRQ* or *PCCol* they then take control of their respective arbiters. The pending event (change in level) is propagated to the arbiter output (*g2*). This output is wired directly back to the "done" input of the arbiter (*d2*) so the arbiter is immediately released again. The net effect of doing this is that the change in level of the input of the arbiter has now propagated to the output. When the next instruction arrives at primary decode it waits for all three arbiters to return to locked (signalled by *Lreq*). The primary decode can then inspect *SFIQ, SIRQ* & *SPCCol* to see if their levels indicate that special action is required. The values are guaranteed to be stable because all the arbiters are locked so any further asynchronous transitions on *FIQ, IRQ* or *PCCol* cannot propagate and affect the primary decode. *SFIQ, SIRQ* and *SPCCol* are stable copies of the inputs which are only allowed to change at a well defined time (between *Pack* and the next instruction arriving), this is shown in figure 4-14. (The external FIQ and IRQ pins are defined to be active Low; the polarity of the interrupt signals shown in the diagrams are active High because an inversion takes place on chip between the pin and arbiters).

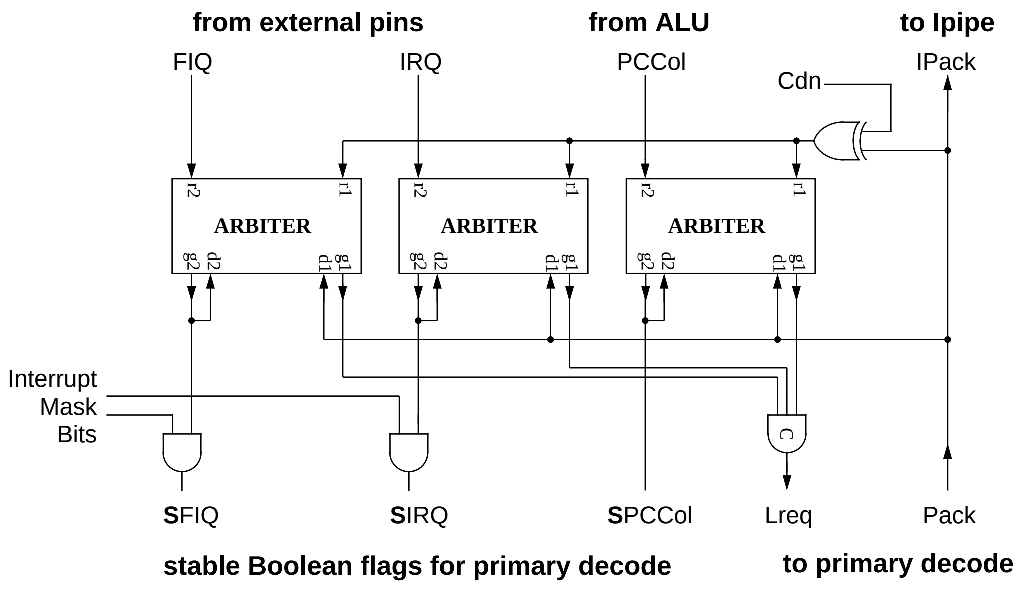


Figure 4-13: Interrupt and PC colour entry into the instruction stream

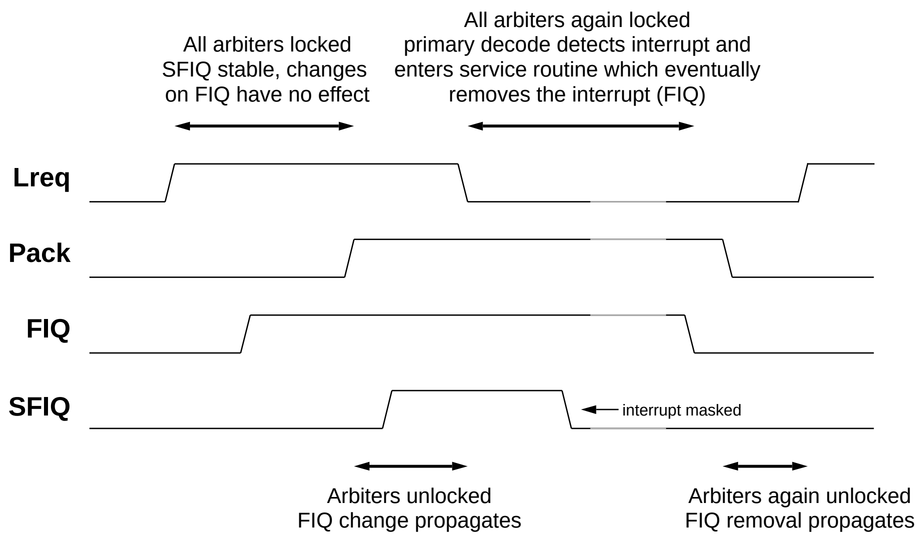


Figure 4-14: Hardware interrupt timing

When the primary decode detects an interrupt request from *FIQ* or *IRQ* the instruction which was about to start decoding is usurped and the exception entry routine is initiated instead. The PC value at the end of the *PC pipe* (and hence the value appearing as *R15*) is the address of the instruction that was usurped (+8). The *R15* value can be read by the exception entry routine and used to form the interrupt return address.

The interrupt flags (*SFIQ, SIRQ*) remain active until they are gated out by writing to the interrupt mask bits in the CPSR. This happens during exception entry (otherwise the processor would repeatedly enter the exception handler).

The exception entry routine inherits the colour of the instruction it usurped. Therefore if the exception entry routine is following a branch which is taken it will be discarded because it has the wrong colour. The interrupt flags remain active so the next instruction which passes the arbiters also gets usurped for exception entry and is also subsequently discarded. This continues until an instruction from the branch target arrives with the correct colour. This time the exception entry routine is successfully executed and the active interrupt flag is masked out.

The reason for not forcing the execution of exception entry cycles which have the wrong colour is that the PC value at the end of the PC pipe associated with the usurped instruction will not give the correct restart address. The correct return address is the branch target. This is automatically delivered by waiting for an instruction with the correct colour.

4.5.2 Data abort overview

Data aborts are by far the most complex exception to handle. The ARM architecture specifies that the instruction which caused the abort must be restartable once the cause of the abort has been removed (e.g. a virtual memory page fault has been fixed). To restart the instruction, the processor needs to determine the address which faulted and the address of the data transfer instruction that issued the faulting address; it must also ensure that its state is preserved whenever an abort is taken (so that a restarted instruction resumes execution in the same environment). An overview of the data abort process is presented here, more detail is provided in the next section.

When a data transfer instruction begins execution, the PC value at the end of the PC pipe is copied into a special FIFO referred to as the exception pipeline (*Xpipe*). When the data transfer reaches the ALU to perform the address calculation, its corresponding PC value is at the end of the *Xpipe*. The ALU sends the data address to memory (via the address interface - see section 4.3.4) and then waits for a response from the memory management unit (MMU) to indicate whether the transfer was successful.

If it was successful, the ALU can continue with the next instruction and the PC value in the *Xpipe* is discarded. If a data abort is signalled then the PC value at the end of the *Xpipe* is copied into a holding latch (*XLatch* - see section 6.1.6) and the exception entry routine entered by signalling to the primary decode. The abort entry routine can then access the exception PC address and store it as the exception return address. To ensure the instruction can restart, the base register (the address of the transfer) must also be preserved. Any base modifications (due to auto-indexing) are allowed to complete because these are deterministic and can be reversed by the exception handling software. Loads which overwrite the base register cannot be permitted to complete because this would destroy the transfer address and hence the instruction could not be restarted.

A single register data load operation, which loads data into the base register, cannot destroy the base if it aborts because no data is retrieved from memory so there is nothing to overwrite the base register with. However, for a load multiple operation it is possible to overwrite the base register early in the sequence and have a subsequent address cause an abort. In this case special measures are needed to ensure that a copy of the base is preserved and is available to restart the instruction (see section 4.3.6).

4.5.3 Data abort signalling

When a data abort is signalled to the ALU, the primary decode must be directed to enter the exception routine. The ALU achieves this by first changing the reference colour so that following instructions are discarded (and therefore do not change the processor state). The change in instruction colour is signalled to the primary decode via the previously described arbiter. This causes all remaining prefetched instructions to be discarded (see figure 4-15) via **SELECT** *S1* and **XOR** *X1*. Once the colour mismatch is established no further instructions can reach the primary decode.

An event is generated when the exception PC holding latch (*XLatch*) becomes full (which must be the result of a data abort). This event is asynchronous with respect to the primary decode and would normally require an arbiter to introduce it into the logic. However, once the PC colour mismatch is detected there can be no further events in the primary decode because incoming instructions are being discarded). The data abort event is therefore safely allowed to pass directly to the primary decode to start the exception entry routine. This safety is ensured because the data abort event is blocked by a transparent latch until the colour mismatch has definitely been established (and hence no more instructions can enter primary decode).

The event to the primary decode is accompanied by a Boolean flag (*Dabt*) to indicate that this is a data abort request. The exception entry routine begins, and saves the exception PC etc. (see section 4.3.7). The various cycles of the data abort entry are forced to execute

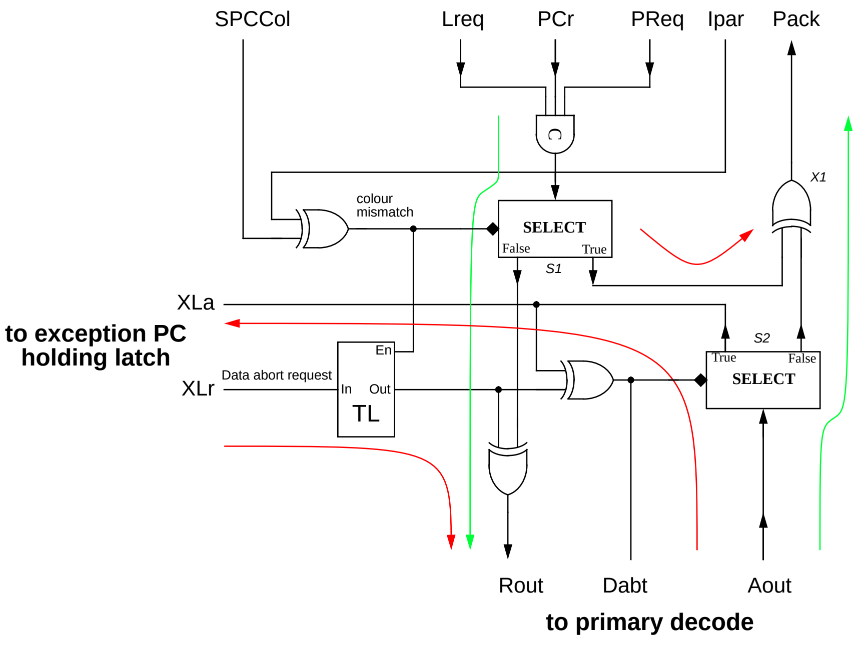


Figure 4-15: Data abort entry into the primary decode

irrespective of the condition codes or colour checking. This is because the data abort entry is not linked to an instruction in the instruction pipeline and does not use an associated PC value; it uses its own value from the *X-Latch*.

When the exception entry is complete the primary decode acknowledges back (*Pack*). This event is steered back to the exception PC holding latch and it also resets the state of SELECT (*S2*) so that it steers subsequent acknowledges back via *X1* to the arbiters and the *Ipipe*. The normal instruction event path is shown in green in figure 4-15, and the data abort event path is shown in red.

Eventually instructions from the exception handler arrive with the correct colour and begin execution as normal instructions.

Chapter 5 : The register bank

A high performance register bank is a central component of a RISC processor. The challenge in an asynchronous design is to maintain coherent register operation while allowing concurrent read and write access with arbitrary timing and dependencies between them. The solution adopted in AMULET1 includes a novel arbiter-free register locking mechanism (described later) which enables efficient read operations in the presence of multiple pending writes.

The special requirements for a register bank in a Micropipelined processor can be specified only after analysing the general operation of a register bank and the hazards which are introduced by a Micropipelined execute unit.

5.1 Register bank operation

A register bank consists of a number of registers with common sets of read and write buses. All registers are the same width; in the 32-bit ARM6 microprocessor they are all 32 bits wide. The microprocessor executes instructions as follows:

The instruction decoder extracts the addresses of the registers to be read (*a, b)* and the register where the result will be written (*w*) from the op-code; these are passed to the register bank to begin execution. The operands are read out and passed to the execution unit where the operation is performed to provide the result. Some time later the result arrives back at the register bank and is written to the appropriate register (*w*). Figure 5-1 shows a

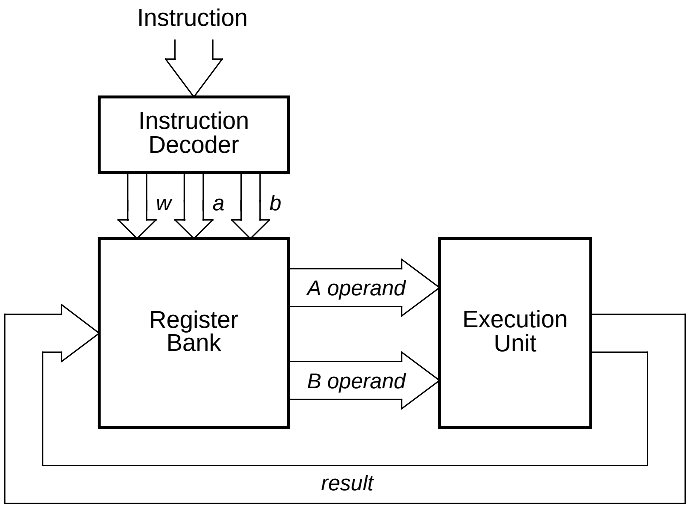


Figure 5-1: Register bank operation

high level view of this process. The number of registers read by any instruction is limited by the number of output ports on the register bank, which in the case of the ARM6 is two.

5.1.1 Internal register structure

Figure 5-2 shows the internal structure of a typical register bank for reading one operand, each register occupies a vertical slice of the register bank. A register comprises of a number of individual memory cells arranged in a vertical stack (32 memory cells for a 32-bit register). Each of the memory cells has an input (which is used for changing its value) and two outputs (which are used to read the value from the cell onto the *A* operand and *B* operand buses). All operations are performed on all the cells of a particular register at the same time. Three enable signals are provided for each register to control reading onto the *A* bus, reading onto the *B* bus and writing from the result bus into the cell.

The *A* bus decoder takes the binary representation of the *A* operand register number (*a*) and converts this into a unary representation (1 out of *N*) used as the *A* bus output-enable for the selected register. Figure 5-2 shows 1 out of *N* registers selected in this way. Similar decoders are provided for the *B* bus and for the write bus which have been omitted from the diagram for clarity.

The *A* operand, *B* operand and write buses all run horizontally through all the registers as shown above; again the *B* and write buses have been omitted for clarity.

At a slightly higher level this can be represented as in figure 5-3. Physically, the decoders are stacked on top of each other with the select wires routed vertically through to the register bank.

5.1.2 Register bank hazards

If the execution phase of the instruction is pipelined then it is possible to have multiple operations in progress at the same time implying that there may be more than one write operation outstanding. To ensure the correct result is written to the appropriate register, a record of destination register addresses must be maintained; this allows a returning result to pair with the correct destination address and to update the correct register.

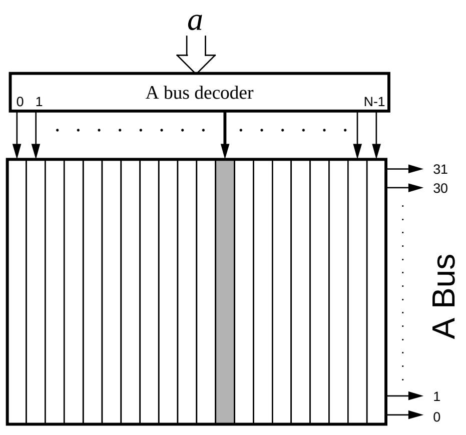


Figure 5-2: Register bank internal structure

Another consequence of a pipelined execution unit is that an instruction may try to read the contents of a register that is to be written by an instruction already in progress. This situation can be caused by an instruction which uses the result of the preceding instruction. If the result has not yet been calculated then the previous value could be read erroneously. To ensure correct operation, the dependencies between instructions must be managed so that reads from registers which have not yet been written wait until the contents are valid.

The third hazard of register bank operation is related to the asynchronous nature of the write operation in relation to a read. The execution unit can take a data dependent time to complete its operation, so there is no fixed time at which a result is expected to return to the register bank. The next read may start as soon as the operand data is latched into the execute pipeline. Therefore reading and writing are asynchronous operations and care must be taken to ensure that any interaction between the two, for example writing and reading the same register, does not cause metastability problems (e.g. when a register value is changed part way through a read operation).

In summary, the three register hazards associated with an asynchronous pipelined processor are as follows:

1. There may be multiple outstanding write operations
2. A read may be requested from a register whose contents are invalid pending a write.
3. Asynchronous read and write operations using the same register may interact unpredictably.

5.2 Write address storage

The obvious way to store the write addresses is in a simple FIFO. Write addresses are put into the FIFO after the instruction reads its operands and removed from the FIFO when the destination address is required to write the result back into the register bank. A standard

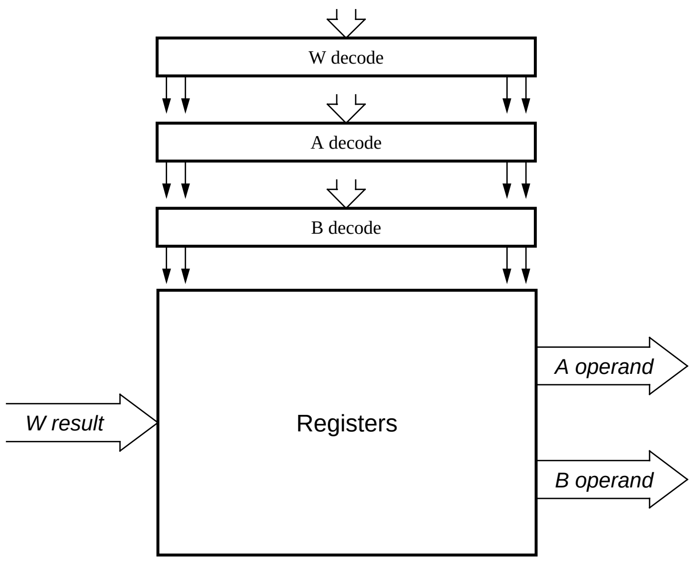
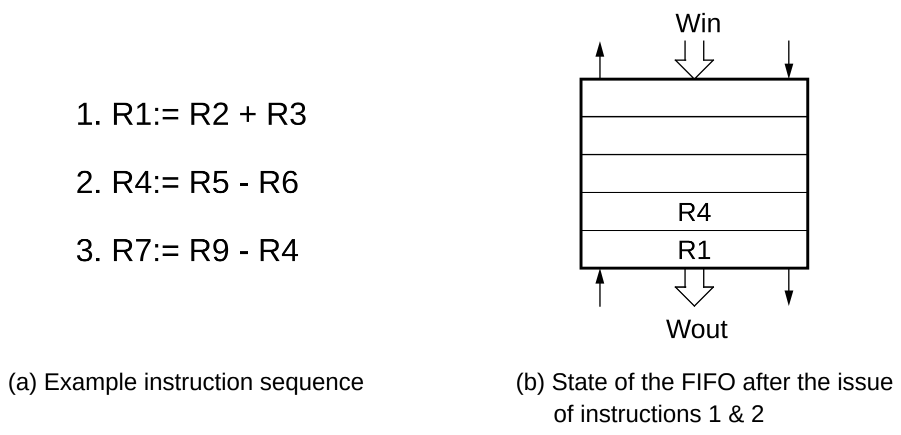


Figure 5-3: Decoder arrangement



(a) Example instruction sequence

(b) State of the FIFO after the issue of instructions 1 & 2

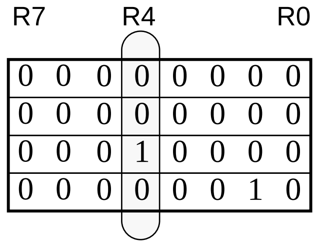
Figure 5-4: Write address storage

Micropipeline provides a simple and efficient way of doing this. Figure 5-4 shows an example instruction stream and how the write addresses are stored within such a FIFO. The example shows that the first two instructions have begun execution and placed their destination write addresses in the FIFO. The third instruction cannot commence until the result of the previous instruction (*R4*) has been written back to the register bank since it is required as one of the operands. The FIFO contains a list of all registers whose contents are invalid. These invalidated registers must be *locked* so that reads to them cannot proceed until the contents are valid. A read operation could therefore inspect the so called “lock FIFO” to determine whether it may proceed or whether it must wait for a value to return. This poses the problem of finding and comparing register values in the asynchronous lock FIFO to determine if the registers to be read are locked. The condition detection mechanism[Pave91] provides a simple and efficient way of achieving this.

5.2.1 An asynchronous register lock FIFO

If the lock FIFO is constructed so that the locked registers are represented as decoded unary values, (i.e. 1 out of N), then each lock entry will have exactly 1 set bit (in the position that corresponds to the register that is locked). For example if there are 32 registers then there will be 32 bits, and if bit 3 is set then this indicates that register 3 is locked pending a write operation.

Using this representation, the determination of whether a register is locked is a matter of establishing whether there is a bit set in a particular column. Figure 5-5 illustrates how the



R7			R4				R0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0

Figure 5-5: Lock detection

FIFO is interrogated to determine if *R4* is locked and whether the read may proceed or not (the FIFO illustrated contains the unary encoded values of *R1* and *R4*). In this case *R4* is expecting write data so the read must stall.

A further, beneficial, consequence of storing the write address in unary form is that the last stage of the FIFO can be used to drive the write word lines of the register bank without further decoding.

5.2.2 FIFO examination

The columns in the lock FIFO can be examined by simply **OR**ing together successive bits of the FIFO as shown below in figure 5-6. This gives two outputs, one that indicates that the next write operation is to this register and the other indicates that this register has a write pending and is therefore locked. For this simple **OR**ing technique to work in an asynchronous FIFO, three conditions must be met:

1. Empty stages must not interfere with the lock and so must present a zero output.
2. Data must be copied to the next stage before being deleted from the current stage to prevent it from transiently disappearing from the **OR** chain.
3. Data must not be allowed to enter the FIFO while the "*Locked*" output is being examined to prevent asynchronous interaction problems (metastability etc.).

Micropipelines, in general, are transparent when empty, so ensuring empty stages give a zero output is a simple matter of ensuring that the input to the FIFO is held at zero while data is not being entered. Another feature of standard Micropipelines is that they automatically copy data to the next stage before deleting it from the current stage, so the second condition is also satisfied.

Instructions which read and write the same register must not lock the destination until the read is complete otherwise they would stall on their own lock. This requires sequential ordering of the read and lock operations. This high-level control constraint also solves the

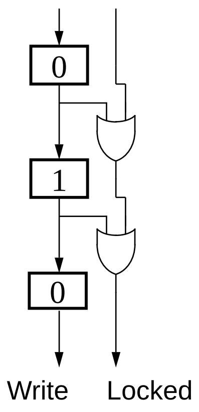


Figure 5-6: Lock interrogation

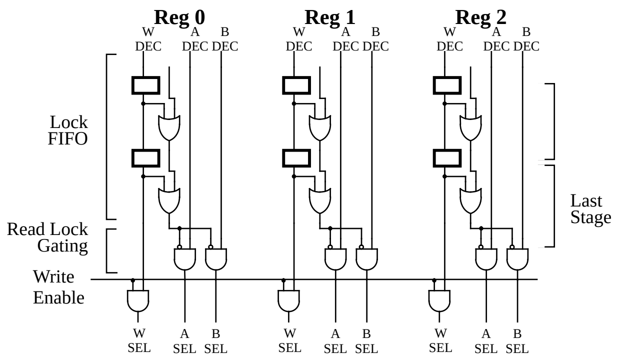


Figure 5-7: Read lock gating

third condition required for correct operation of the lock FIFO. The detail of the high-level control is discussed later.

5.2.3 Stalling reads

The lock FIFO provides a mechanism which indicates that a register has a write pending. This information must be used to prevent any read operations from proceeding until the data has returned and the write operation is complete. Figure 5-7 shows how the lock FIFO *locked* output and the *A* and *B* decode word lines are combined with a simple **AND** gate to stall reads. A lock in any column effectively disables the **AND** gates for that register and so prevents the register read from enabling the contents of the locked register onto the operand buses. A read will remain stalled until the lock entry reaches the last stage and is matched with the corresponding write data. The write operation then completes and the bottom entry of the FIFO is removed; the corresponding set bit will then disappear and allow the stalled read operation to proceed.

The diagram also shows how the last stage of the FIFO is enabled onto the register bank write word lines (*W Sel*) under the control of a write enable signal.

5.3 Asynchronous register bank design

The overall organization of the register bank is shown in figure 5-8 [Pave92a]. The interfaces use the bundled-data convention with transition signalling. Internally, the design employs a combination of two-phase and four-phase techniques, the latter being well matched to the precharge-active cycle of the dynamic circuits used in the basic register cell.

5.3.1 Read and lock operations

A new instruction has its availability signalled by *I-Req*, and presents two register addresses to be read (*a* and *b*) and a register address to be written (*w*) once the execute unit result is available. *I-Req* is stalled until the register bank is ready to start a new read operation when the read decoders are enabled. Concurrently with the read address decoding, the write register address is latched (in *W Latch*).

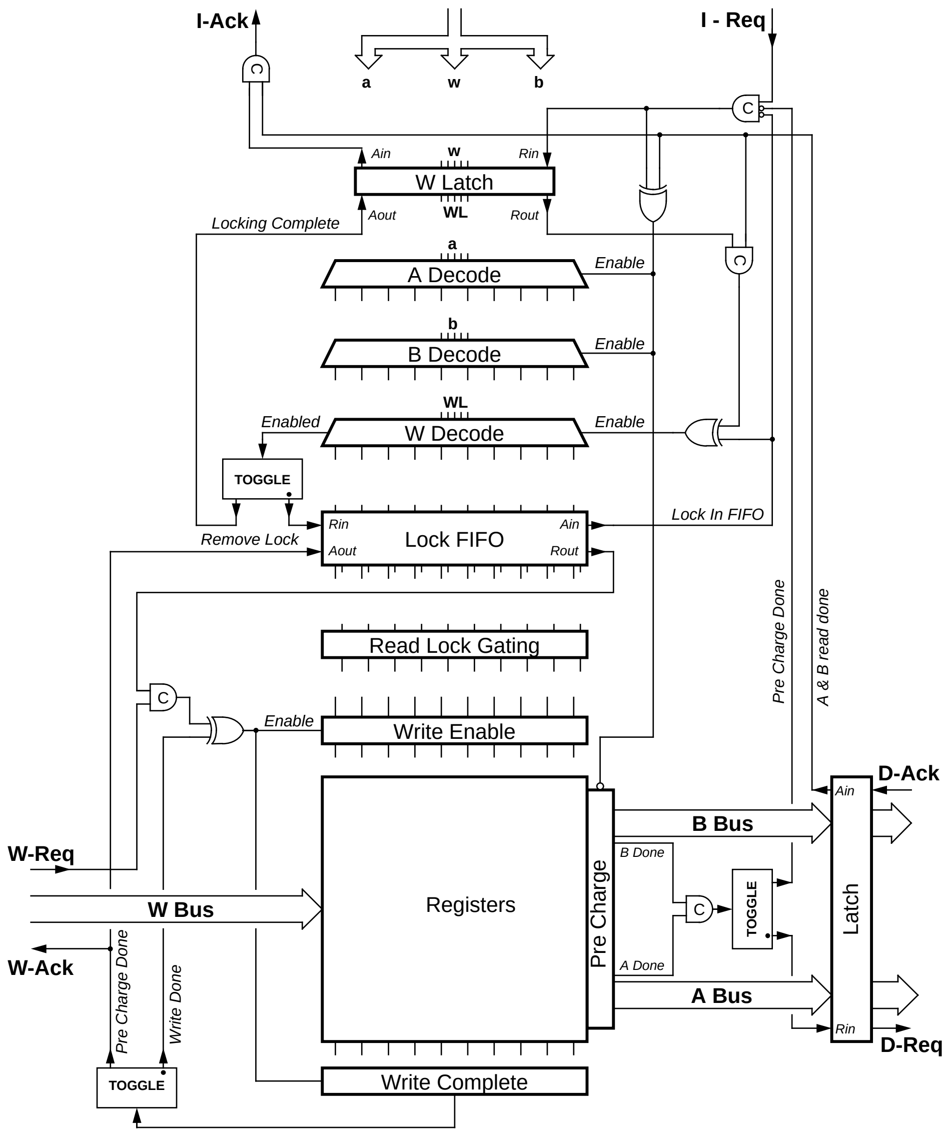


Figure 5-8: Register bank organization

The decoded read addresses present "enables" for the selected registers which are gated with the locked register information. A read of an unlocked register will proceed, whereas a read of a locked register will stall at this point until a write operation clears the register lock.

The register read circuitry uses dynamic techniques to minimise the cell size, with charge retention circuits to give pseudo-static operation. A extra thirty-third bit line gives a matched completion signal and when both register values are available they are latched and passed to the execution path (via the *D-Req* signal) which can begin to process the data with no further delay.

Once the data has been latched, the read decoders are disabled and the read bus precharge is turned on to prepare for the next access. Normally the write address will be latched well

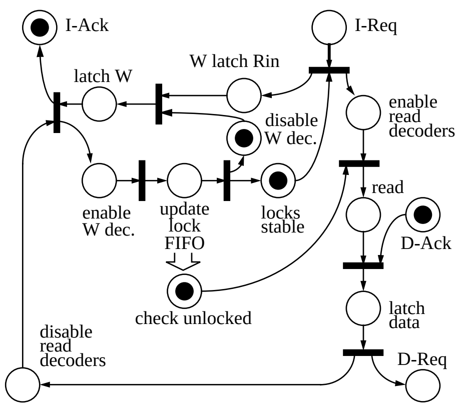


Figure 5-9: A Petri Net model of the read-lock sequencing

before this time, and the instruction acknowledge (*I-Ack*) is issued so that a new instruction can be prepared during the register recovery time.

The write decoder is disabled during the read operation to present inactive inputs to the lock FIFO. Once the read data is latched, the write decoder is enabled and the destination register is locked. As soon as the lock FIFO has accepted the new address, a new instruction may be allowed to start its read operation. The lock logic will continue by disabling the write decoder, it will then free the write address latch for the next value.

The read-lock sequencing is illustrated in Petri Net form [Pete81] in figure 5-9, which shows the critical sequential dependencies in the read operation. Note particularly that the *W* decoder is disabled until the read has completed, in order to ensure that no spurious lock indications are passed via the empty (and therefore transparent) stages in the lock FIFO; similarly it is disabled before the *W latch* is allowed to accept a new value. The next read is allowed to proceed as soon as the locks are stable, since any transient caused by the slow disabling of the *W* decoder will cause at worst a delay in the read operation, never an incorrect action.

The critical path in the register bank (from *I-Req* to *D-Req*) has the minimum number of dependencies on internal operations; this defines the register access latency of the design. The cycle time will include this and the slowest of three independent recovery routes:

- The supply of the next instruction.
- The completion of the locking operation.
- The read bus precharge time
  (omitted from figure 5-9 for clarity).

In general it is expected that the first of the above will be the critical path in determining the register bank cycle time.

5.3.2 Write operations

A write data value (signalled on *W-Req* in figure 5-8) is paired with the decoded write address at the output of the lock FIFO. The appropriate write word line is then enabled and the data written, following which the destination register is unlocked for reading by removing its address from the FIFO. The write operation is self-timed by detecting the transitions on the word line with a wide dynamic OR gate, the same circuit ensures that writes are fully disabled before the write data is allowed to change

5.4 Additional features

There are several features described in this section which were omitted from figure 5-8 to maintain the clarity of the diagram. The write operation includes a Boolean flag to indicate whether the data is valid. This allows instructions which have failed condition tests at the ALU to return to the register bank to remove locks placed previously, without writing any data. The valid flag is used to steer the write request to remove the last item in the lock FIFO without enabling the write word lines. No data is written into the register bank.

The program counter on the ARM is available as a general purpose register (*R15*). The value of the program counter for any particular instruction arrives with the instruction (from the PC pipe) so the register bank simply transfers the value onto the operand bus if required. The multiplexer control is similar to the read control of the general purpose registers; however, as *R15* is not an actual register in the bank, there is no notion of writing to *R15* within the register bank. Therefore there is no lock FIFO entry for *R15* and reads from *R15* can never be stalled on a lock. The value of *R15* is changed by sending the new value to the address interface (a branch operation).

Some instructions do not require all three register addresses (*a*, *b* & *w*) so logic is supplied to bypass a subset of them. To economise on logic for the read operations, instructions which do not need both operands read the PC value instead and then discard it. It is safe to do this because reads from *R15* can never stall.

The register bank also contains several special registers which contain saved versions of the processor status register (*SPSR*); these can be accessed only via the *A* operand bus but are enabled in a similar manner to the general purpose registers. Extra information is provided with the instruction to indicate that the access is to an *SPSR*. Write operations to an *SPSR* can specify that only part of the status word is to be changed (e.g. arithmetic flags, control flags or both), extra information therefore is required for write operations to an *SPSR* to indicate which parts are to change. When a write to an *SPSR* is detected the extra information is stored in a control FIFO and maintained in step with the main lock FIFO; when the write address for the *SPSR* reaches the bottom of the main lock FIFO, the corresponding information in the control FIFO is therefore also at the bottom. This is then used to control the write enable of the two halves of the *SPSR* depending on what the instruction specified.

5.4.1 Dual lock FIFO

Instead of a single lock FIFO as described earlier the implementation has two FIFOs; one for operations that load data from memory and one for internal operations where the data comes from the ALU.

The dual FIFOs allow internal ALU cycles to overtake slower external memory accesses assuming there are no register dependencies between the two. This gives rise to potential compiler code-reordering optimizations to reduce dependencies on loaded data and to increase performance (independent ALU cycles can be executed while the data is being fetched).

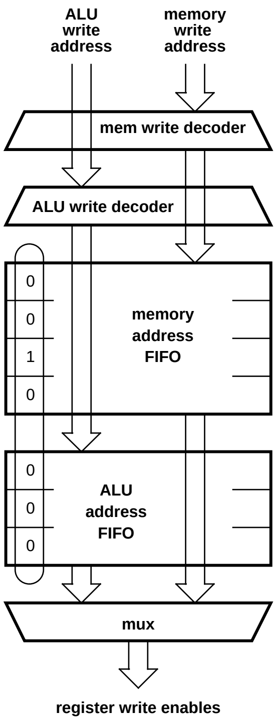


Figure 5-10: Dual lock FIFO configuration

Figure 5-10 shows how the two FIFOs are configured so that they can still correctly indicate a locked condition by **OR**ing across both FIFOs. Two decoders are provided, one for each FIFO and the primary decode supplies two write addresses. This allows an entry to be placed in both FIFOs concurrently and is used to support instructions which have two destination registers (e.g. load data with base writeback; here the two registers are for the data coming back from memory and for the modified base value returning from the ALU). If only one destination address is needed, only one of the write decoders is enabled, and only the appropriate FIFO activated. Once the destination address is in the correct FIFO, the write decode is disabled and the operation proceeds as before.

When a *W-Req* is received it is accompanied by a Boolean flag to indicate whether the data value to be written came from the ALU or memory. This is used to multiplex the correct write destination address onto the write word lines before the write commences. When the write is complete the last entry in the selected FIFO is removed and any read stalled on this write is released automatically.

5.5 Implementation

Storing the write addresses in the lock FIFO in unary encoded form may appear inefficient in the use of silicon area, but it allows the full stack of word control logic (the *A*, *B* and *W* decoders, the lock FIFO, read lock gating and the write enable logic) to be pitch-matched to the register cell block. Figure 5-11 shows the silicon layout of the register bank. The lock FIFO can clearly be seen as the dense regular layout above the datapath register cells. The *A*

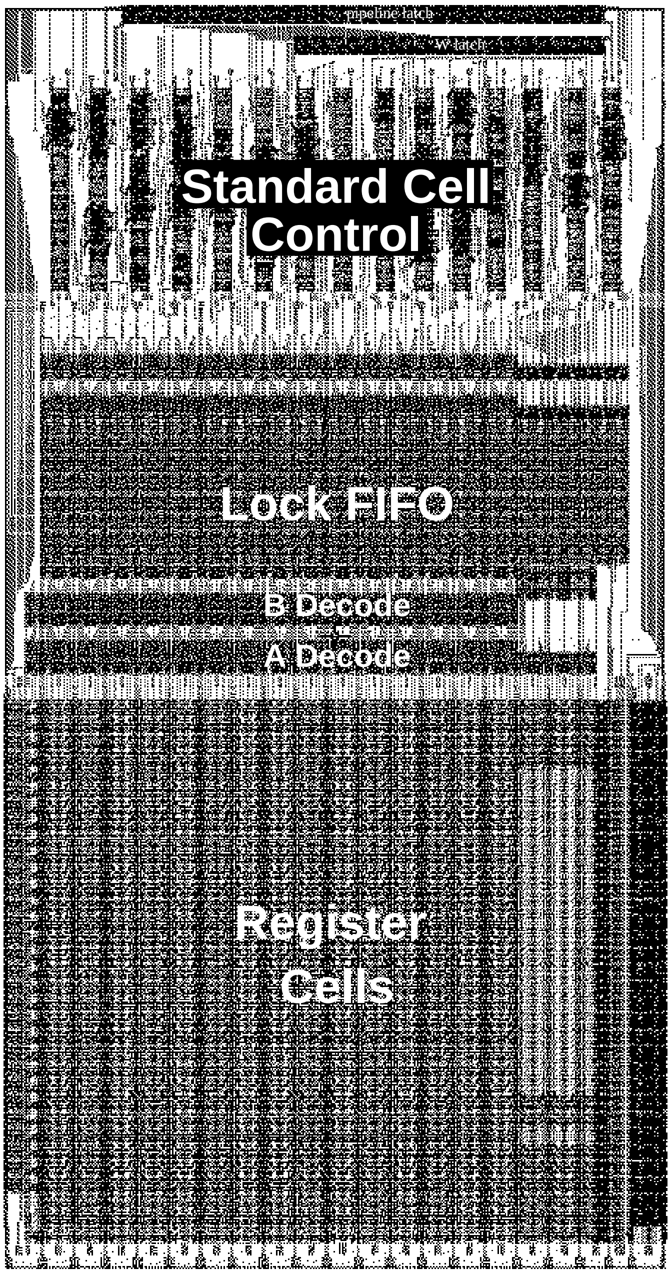


Figure 5-11: Register bank silicon implementation

and *B* decoders (labelled in figure 5-11) below the lock FIFO have the read lock gating built into them with the “locked” signal provided from the FIFO above.

The main core of the bank - the datapath register cells - are identical to those used in the ARM6 with bit 31 at the top and bit 0 towards the bottom. The thirty-third bit used for the self-timing path is in an extra row at the bottom just above the power rails. Using the existing ARM6 register cells maintains the area efficiency of the original synchronous datapath area and illustrates the ability of the Micropipeline design methodology to re-use synchronous elements.

Figure 5-11 also features the 5 *SPSRs* which are the sparse registers to the right of the datapath cells. The sparse nature results from the fact that the processor status word only contains information in bits 31-28, 7, 6 and 4-0 so storage in the other bit positions is not needed (see appendix B for information about the PSR format).

The register bank control logic is implemented in the automatically compiled area of standard cells shown towards the top of the diagram. The decode pipeline latch and the register bank *W latch* can be seen as the two thin horizontal strips at the extreme top of the diagram.

5.6 Future enhancements - register bypassing

Typical instruction streams frequently display the use of the result of one instruction as an operand of the next. Such data dependencies between consecutive instructions can cause a significant reduction in throughput for typical code, compared with best case code without dependencies, if the result is only available to the next instruction after it has been written back to the register bank.

Clocked processors generally use *register bypassing* to allow a result to be re-used without incurring the register write-then-read penalty. The global clock ensures that different parts of the processor are operating at fixed relative times, so the result and operand addresses at two stages can be compared to activate the bypass when appropriate.

In an asynchronous processor there is no such fixed relationship between the timing of operations in different parts of the processor, so explicit synchronisation is necessary if a similar result and operand address comparison is to form the basis of a bypass mechanism. This synchronisation will have a cost in reduced throughput and, since it forces lock-step operation of at least two parts of the processor, is a serious obstacle to fully exploiting the advantages of asynchronous operation.

Two alternatives to register bypassing have been considered which deliver some of the benefits without impeding the asynchronous operation of the pipeline. Although these have not been implemented in the current design, they are included as suggestions for future enhancements and are described below as register through-passing and last result re-use.

5.6.1 Register through-passing

The design shown in figure 5-8 is very conservative in its timing for a write operation which clears a lock and thereby allows a read to proceed. A mechanism under consideration would, with the addition of a latch to the write enable logic, allow the lock to be cleared much earlier in the write process (see figure 5-12). As soon as the write is complete the lock is removed from the FIFO to enable the read to proceed while the write enable lines are held stable by the *write enable latch* until the write enables are safely turned off. This prevents writes to spurious registers.

5.6.2 Last result re-use

Another mechanism under consideration detects data dependencies at the decode stage. Each instruction leaves behind in the instruction decoder a record of its destination register, when the next instruction enters the decoder its operand addresses are compared with this record. If a match is found, the read operation is bypassed and the result is collected for operand use directly. The mechanism has no effect on the design of the register bank in figure 5-8 as it is manifested in additional logic elsewhere in the decode and execution paths.

This second mechanism has better performance than the first when it is applicable, but it has several limitations. A particular problem on the ARM is that all instructions are executed conditionally so an instruction which fails to pass the condition test will not produce a result. By this time its successor may depend on that result, so this mechanism must include logic to determine whether or not an instruction may be annulled (and hence will not produce a result). This adds to the complexity of the design.

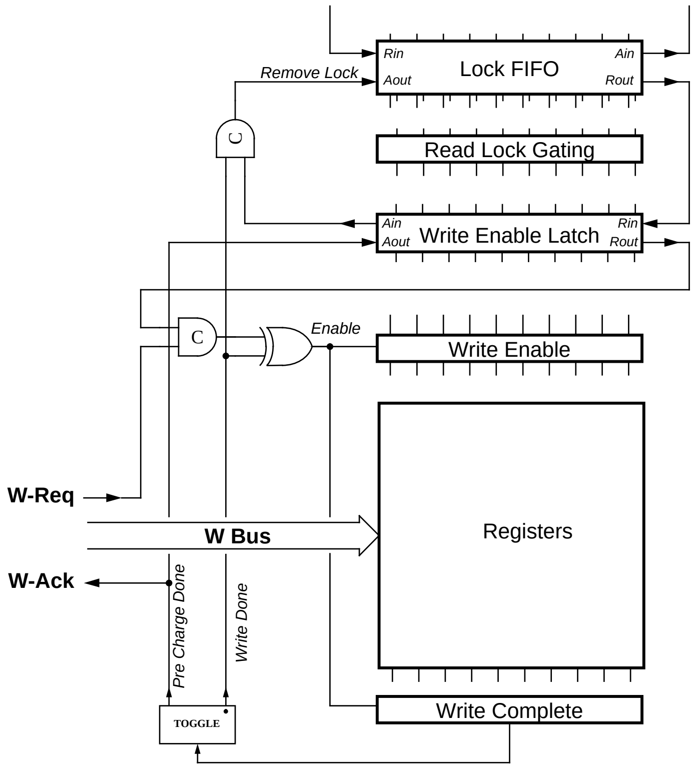


Figure 5-12: Improved register write circuitry

Chapter 6 : Memory interface

The memory interface can be divided into two distinct parts: the address interface, responsible for issuing all address information to memory, and the data interface which manages the flow of data into and out of the processor.

A memory access can be either a read or a write operation. For a write operation, the address supplied by the address interface synchronizes with the data supplied by *data out* before being dispatched to memory. For a read operation, there is no immediate need to synchronize with the data interface when issuing the memory address. However, when the read value returns, the data interface must know whether the value returned is read data destined for the register bank or an instruction to be queued ready for execution.

To enable the data interface to route incoming values to their correct destination, a Micropipeline FIFO containing control information is connected between the address and data interface. For every read request issued by the address interface control information is placed in this *memory control FIFO*. Every read value arriving from memory pairs up with its control information taken out of the memory control FIFO and is steered to the correct destination.

6.1 Address interface

The main function of the address interface is to generate sequential instruction addresses. This is achieved by circulating the PC around a loop containing an incrementer (see section 6.1.7 for further information about the incrementer). For each pass around the loop (shown in figure 6-1(a)) the next sequential address is sent to memory.

After reset, processor execution begins with the memory address register (MAR) being forced to all zeros and an event being generated to start the processor prefetching. The first value sent to memory is *0*. This value is also passed to the incrementer where the next instruction address is generated (PC +4). The incrementer is a dynamic structure so the result is stored in the PC holding registers - *PC HLS* (the reason why two registers are needed is discussed later). If no other address source wishes to use the address interface, then the PC value returns to the MAR via the multiplexers. The next instruction address is dispatched to memory and the process repeats.

The program counter can circulate around this loop at its own speed, decoupled from the actions in the rest of the processor. If the processor wishes to use the address interface to generate the address for a data transfer the PC incrementing loop must be temporarily interrupted; the PC loop is asynchronous to the rest of the processor, so an arbiter is required to manage the interaction. The data transfer begins with the transfer address being supplied on either the write bus or the *A* bus (depending on the instruction and whether pre- or post-indexing is specified - see section 4.3 for details of how addresses arrive at the address

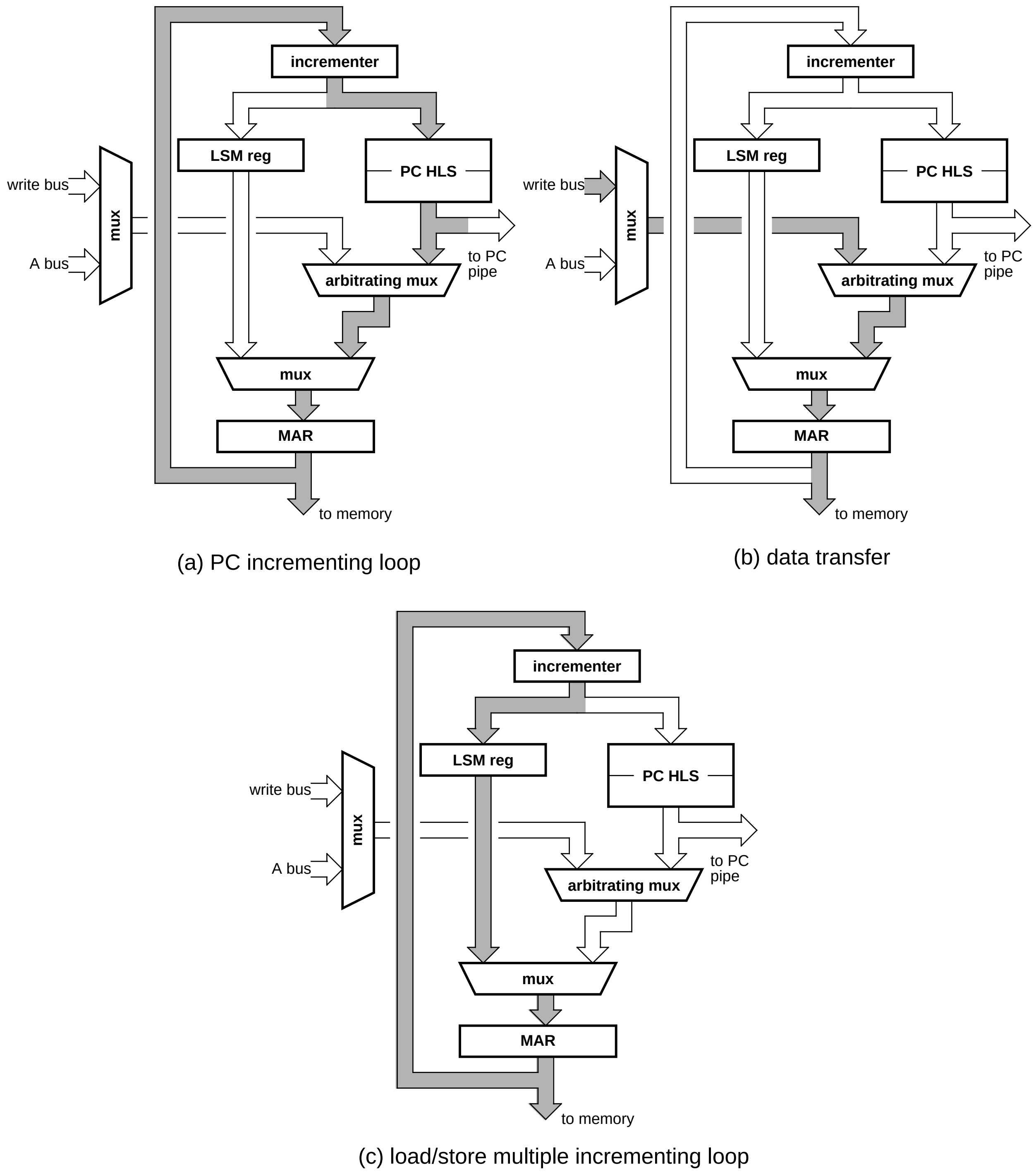


Figure 6-1: The address interface

interface). Arbitration takes place between the PC value and the transfer address to take control of the MAR. Eventually the data transfer takes control and passes its address out to memory via the arbitrated multiplexer and MAR (figure 6-1(b)). Once the transfer is complete the arbiter is released allowing the pending PC to continue to circulate around the loop.

6.1.1 Load/Store multiple operation

A load/store multiple operation sends only the base address of the transfer to the address interface and the sequential transfer addresses are generated by modifying the base using the incrementer (similar to sequential PC address generation). A load/store multiple operation begins with the base address arriving on the write bus in the same way as for a data transfer. Once control is taken of the MAR, the interface waits for a signal from the primary decode. This signal contains information about whether this is the last transfer and for a load operation it indicates whether the data being loaded is the new PC value. The last transfer information is used to stop any further incrementing and if a new PC transfer is signalled then the interface is initialized to expect the new PC value.

Once the signal is received from the primary decode, the address is dispatched to memory; if the transfer is not the last, the address is sent to the incrementer where the next sequential address is generated. The incremented address is stored in the load/store multiple holding register (*LSM reg*).

To take advantage of fast sequential modes of DRAM and some cache memories, the PC is forced to wait until the transfer is complete before prefetching is allowed to continue (this ensures that the LSM addresses are uninterrupted sequential addresses). This is achieved by not releasing the arbiter until the transfer is complete. A consequence of this is that the re-circulating LSM address is not subject to arbitration to take control of the MAR because everything else is locked out of the loop. Therefore, as soon as the *LSM reg* has received the appropriate signal from the primary decode, it can forward its value directly to the MAR as shown in figure 6-1(c). The sequential address generation continues until the primary decode signals that the transfer in progress is the last one. In this case, the address is discarded after the transfer is complete (rather than incrementing it). The arbiter is then released to allow the PC to continue to circulate and issue instruction addresses to memory.

6.1.2 Changing the PC value

The PC value circulating in the address interface has an associated colour related to the main colour mechanism (see section 4.4.3). This is used to discard old circulating PC values when a branch is taken. The address interface control contains a reference colour against which the colour of the circulating PC value is compared; if the colours do not match the PC is removed from the loop and discarded.

When a branch is executed, the new PC value arrives on the write bus from the ALU, bringing with it a copy of the new PC colour. Once the new PC value has taken control of the arbiter, the reference colour is updated to the new value from the ALU. The new PC value then appears on the output of the MAR and is sent to memory and the incrementer.

Once the arbiter is released, the old PC value, which has waited in one of the holding registers, can gain control of the arbiter. As it enters the control circuitry, a colour mismatch is detected and the old PC value is discarded. The new PC value completes circulating the loop and presents a request to the arbiter. This time, when control is granted by the arbiter, the PC colour matches the new reference colour so the PC value is allowed to continue to the MAR unhindered. The PC modification is then complete.

6.1.3 PC loop deadlock

The PC incrementer loop includes two holding latches (*PC HLS* in figure 6-1) to prevent a potential deadlock situation arising as described below:

Consider the case with only one holding latch (*PC HL*); the old PC value is waiting in the holding latch and the new PC already in the MAR as shown in the simplified diagram of the address interface shown in figure 6-2(a).

The new PC value cannot circulate any further until the holding latch is free. In most cases, when the arbiter is released, the old PC immediately gains access and is discarded, freeing up the holding latch and enabling the new PC to continue to circulate.

However, if instead of the old PC value gaining control of the arbiter, a data transfer request takes control, the deadlock situation shown in figure 6-2(b) can occur. Here the data transfer address cannot enter the MAR because the new PC is still occupying it waiting for the PC holding latch to become empty. The holding latch is waiting to gain control of the arbiter and the arbiter will not be freed until the data transfer address is safely in the MAR, hence deadlock results.

If two PC holding latches are provided then the new PC value can move around the loop leaving the MAR clear for any data transfers.

Although data transfer operations immediately following a branch should be discarded because the branch has been taken, a load which is cancelled must still release any destination registers locked previously. To ensure that the write/unlock operations maintain

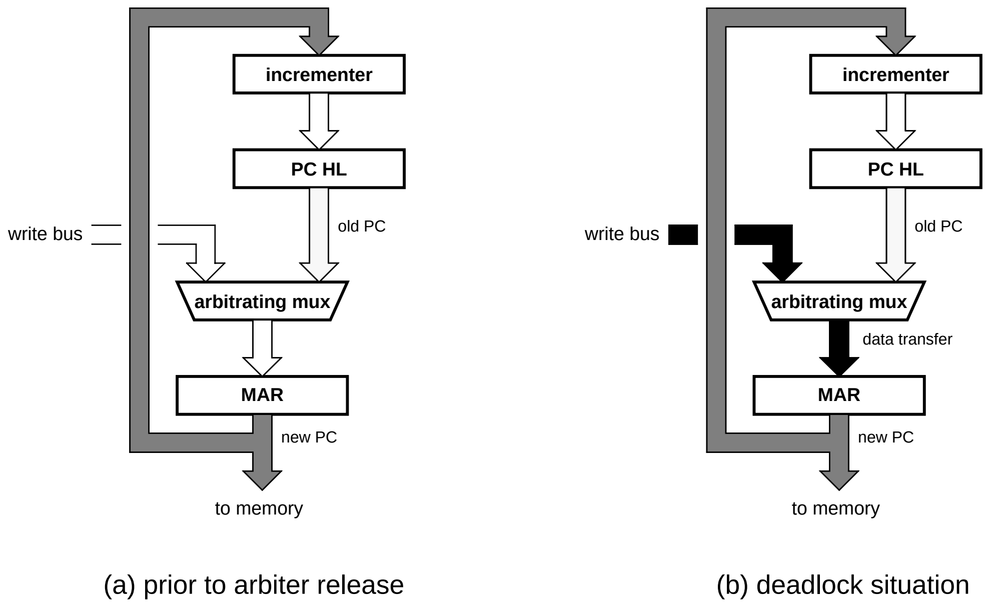


Figure 6-2: PC loop potential deadlock

strict sequential ordering, the unlock request is placed in the memory control FIFO. This is accessed via the address interface so annulled loads still pass through.

When a cancelled load reaches the end of the memory control FIFO in the data interface it is forwarded to the register bank to unlock the destination. Since the load was cancelled, no address was sent to memory and the FIFO entry therefore does not rendezvous with any returning data - it is just sent to the register bank as soon as it reaches the end of the memory control FIFO in the data interface.

The other function the address interface must perform is to supply the PC value to the execution unit for use as the *R15* value. The PC values are stored in a pipeline whose input is connected to the incrementer PC holding latch (as shown in figure 6-1). The operation of the PC pipeline is described below, noting particularly how the value presented as *R15* is controlled to emulate the behaviour of *R15* in the synchronous design.

6.1.4 PC pipeline

As previously described, after hardware reset, the processor starts execution from the bottom of its address space by forcing the MAR to output zero along with the appropriate event control signals. The first value to appear on the output of the incrementer is thus *00000004*. If the output of the incrementer PC holding latch were simply connected to the input of the *PC pipe* then the value PC+4 would be delivered to be synchronized later with the instruction (fetched from address PC). The value actually required for ARM6 backwards compatibility is PC+8. To allow for this, the first PC value after hardware reset is not placed in the *PC pipe*. The first instruction effectively matches up with the “PC+4” of the following instruction hence giving PC+8. This is shown in figure 6-3.

Figure 6-4 shows how the PC values are managed when a branch in the instruction stream occurs; for example, if the instruction at address *4* is an unconditional branch to location *44*. The instructions immediately following the branch (*I3* & *I4*) are assumed to have already been prefetched before the branch instruction was decoded and executed; they must therefore be “thrown away” along with their corresponding PC values. Instruction *I4* is incorrectly matched up with a value from the *PC pipe* (in fact it matches with the PC+4 of the branch target) but this is irrelevant as the instruction is not executed. The branch target instruction is then matched correctly with its PC+8 value.

6.1.5 Instruction overflow deadlock

If no values are removed from the *PC pipe* it will eventually fill; this could happen if the currently executing instruction takes a long time to complete (e.g. a multiplication). When

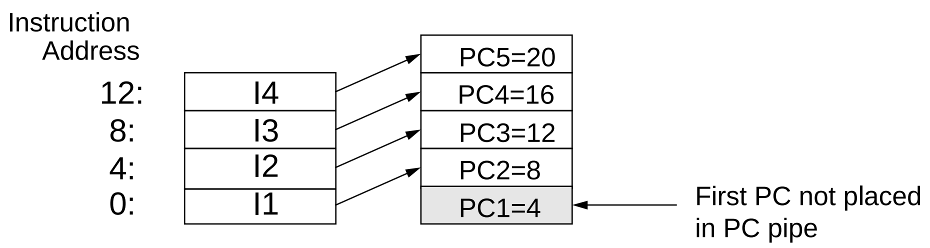


Figure 6-3: Staggering of PC pipe to achieve correct R15 value

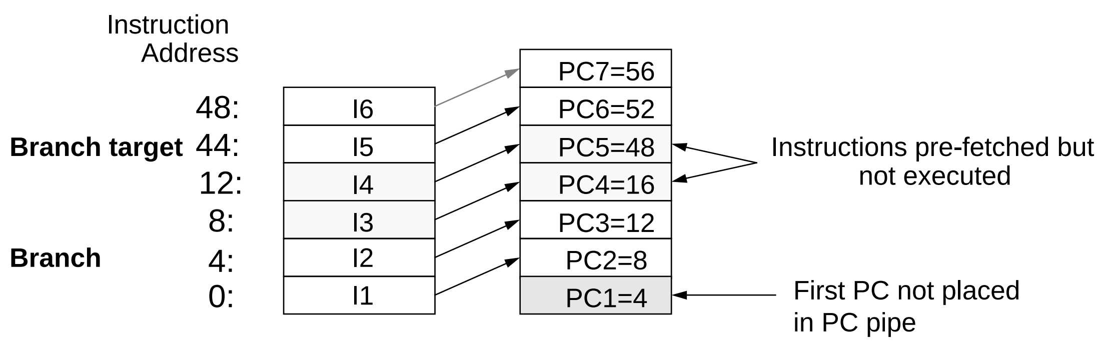


Figure 6-4: PC pipe behaviour across a branch

the *PC pipe* is full, the incrementer PC holding latch will issue a request to it but the *PC pipe* will not acknowledge until there is space in the FIFO for the PC value to be stored. This prevents the incremented PC value reaching the MAR, so no further instruction addresses are issued to memory until the *PC pipe* has some free space.

The length of the PC pipeline determines how many instructions can be outstanding at any particular time, this acts as a self regulating queue or *throttle*. This throttle mechanism is important in preventing a possible deadlock situation as described below.

If the datapath is stalled at the register bank, waiting for a data value to be loaded from memory into a register, instructions will begin to backlog and gradually fill up the instruction FIFO. If there was no throttling mechanism it would be possible for the instructions to backlog all the way into the memory. If the data value happens to be behind this backlog, it would not then be able to complete the transfer to the register bank because the instructions are blocking the memory interface. The instructions are backlogged because of the stall in the register bank which cannot be released because the data value cannot get to the register bank; hence the system is deadlocked. This is shown in figure 6-5.

To ensure that this deadlock can not happen the number of instruction requests issued to memory must not exceed the number of spaces in the instruction FIFO. Thus any backlog is restricted internally to the instruction FIFO itself, leaving the data interface clear to allow the data value to return.

PC pipeline Length

The address interface stops issuing instruction addresses when the *PC pipe* becomes full, this is a function of the number of stages in the *PC pipe*. If the pipe is N stages long then when it is full, N+1 instruction address will have been issued to memory (the first PC value after reset does not enter the *PC pipe*). Therefore the instruction FIFO must be at least N+1 stages long to accommodate all instructions returning from memory, associated with values in the PC pipe. When the *PC pipe* is full, there will also be a PC value in the PC holding latches waiting to enter the *PC pipe*. The instruction corresponding to the PC value in the holding latch will have already been sent to memory, so there are in fact N+2 outstanding instructions which need accommodation in the instruction FIFO (*I pipe*) to prevent deadlock.

If a branch is taken when the old PC value is waiting to enter the *PC pipe* (with N+2 instructions already outstanding) the new PC value arrives from the ALU and is sent out to

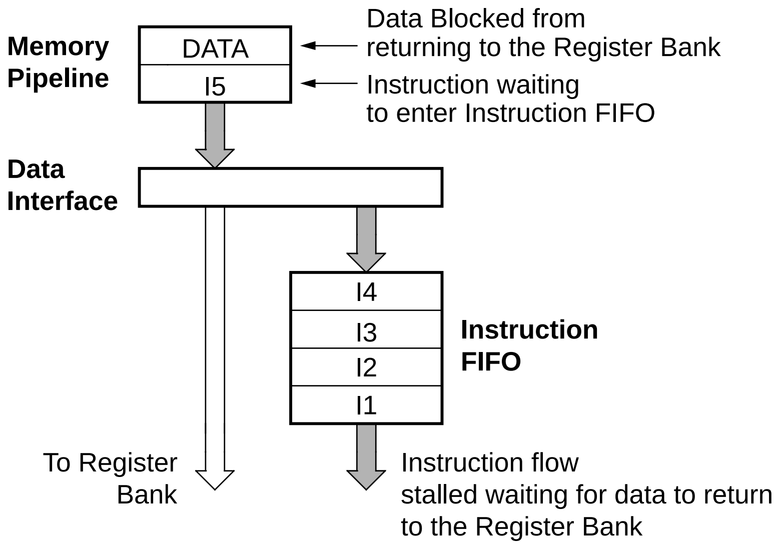

Memory Pipeline
DATA
I5
Data Blocked from returning to the Register Bank
Instruction waiting to enter Instruction FIFO
Data Interface
I4
I3
I2
I1
Instruction FIFO
To Register Bank
Instruction flow stalled waiting for data to return to the Register Bank

Figure 6-5: Potential processor deadlock

memory making the total of outstanding instructions N+3. If a load follows the branch then this is annulled, but must still unlock any destination registers locked previously. To ensure that the write/unlock operations maintain strict sequential ordering, the unlock request is placed in the memory control FIFO. This is accessed via the address interface so annulled loads still pass through.

If the processor is stalled waiting for the destination of the cancelled load to be unlocked, deadlock will ensue if the memory control FIFO is blocked by any outstanding instruction requests, i.e. all outstanding instructions must be in the instruction FIFO. The *IPipe* must therefore be 3 stages longer (N+3 stages) than the *PC pipe* (N stages). This demonstrates the complex relationship between the depth of the PC and instruction pipelines.

6.1.6 PCpipe implementation

The PC pipelines are split into two parts: PC storage for supplying *R15* values (*PC pipe* in figure 6-6) and *Xpipe* for storing the addresses of instructions in progress which could potentially generate a data abort (i.e. all data transfer instructions).

When an instruction begins execution its corresponding PC value is in the last stage of the *PC pipe*; for normal operation this value is selected by the multiplexer and made available as the *R15* value needed by the register bank. Once the primary decode of the instruction has completed, the PC value can be released, if the instruction is not a data transfer the PC value is removed from the *PC pipe* and discarded. If however the instruction is a data transfer then the PC value is placed in the top of the *Xpipe,* and is removed from the bottom only when the corresponding memory access is complete.

Figure 6-7 shows the detailed operation for removing a PC value from the *Xpipe*. Each data transfer has a corresponding PC value in the *Xpipe* which awaits a response from the memory management unit (MMU). There is a single PC value and MMU response for each load/store multiple transfer.

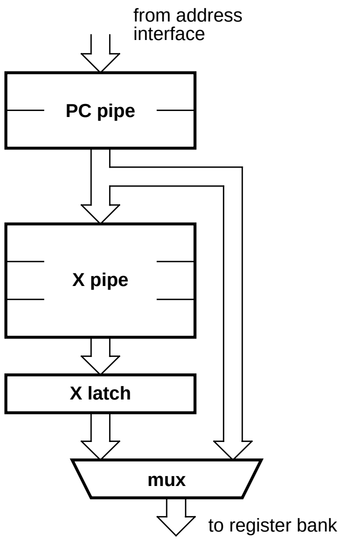


Figure 6-6: The PC pipelines

When the PC value reaches the last stage of the *Xpipe* its request out signal primes a **Decision-Wait** element ready for the MMU response. The MMU issues an event on either *data abort* or *no data abort*. If there is no data abort then the request out of the *Xpipe* is steered back as its own acknowledge (via the **XOR**), effectively removing the PC value in the process.

When a data abort is signalled the last stage of the *Xpipe* (containing the PC+8 of the instruction that aborted) is copied into the exception holding latch (*X-Latch*). The request out of the holding latch is the event used to indicate to the primary decode that a data abort has occurred. The exception entry routine needs access to the address of the failed instruction so that it can retry the instruction when the cause of the exception has been

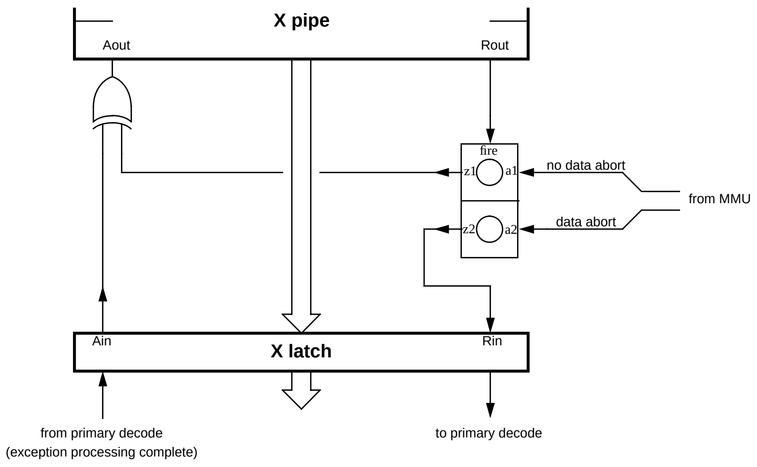


Figure 6-7: Data abort PC storage

removed. The exception address (in *X-Latch*) is multiplexed onto the output of the PC pipelines and is made to appear as *R15* in the register bank, where it can be read and copied into the exception return register, so that when the data abort recovery software routine exits, the processor automatically returns and retries the aborted instruction. Once the exception PC is safely stored elsewhere, the *X-Latch* can be acknowledged and returned to its empty state.

6.1.7 Incrementer

The incrementer is constructed using a simple ripple-carry mechanism with completion detection. Each bit-cell is simple and identical as shown in figure 6-8. The worst case time for this style of incrementer is large because the carry may have to propagate through 30 bits (since PC and LSM are always word aligned there is no need to consider the bottom 2 bits, hence 30 bits worst case). On average, two bits change per operation [Gars92], so the typical case is much faster than the worst. In a synchronous system the cycle time is limited by the worst case, but with an asynchronous implementation the cycle time can vary depending on the data and the extent of carry propagation, so the circuit can be optimized for typical values instead of worst case.

The incrementing operation begins with a carry injected into the lowest bit; if the input to the stage is Low (0) the carry will propagate no further (0+1=1 carry 0). The input selects the path the carry takes through the de-multiplexer; with the input Low (0), the carry is steered to the completion detection circuitry because the increment is finished; if the input is High (1), the carry-in must propagate through the stage when it arrives, to the next highest bit (1+1= 0 carry 1). This is done by the input selecting the other output of the de-multiplexer. The carry continues to propagate until it reaches a stage where the input is Low; here, instead of the carry propagating any further, it is used to signal competition.

The completion circuitry is implemented by a simple wired-OR of the completion signal in every bit. Only one bit will signal completion per incrementing operation. The circuit is implemented using dynamic circuitry to reduce the transistor count to only 20 transistors per bit. Weak-feedback charge retention is provided on the carry inputs to ensure that the

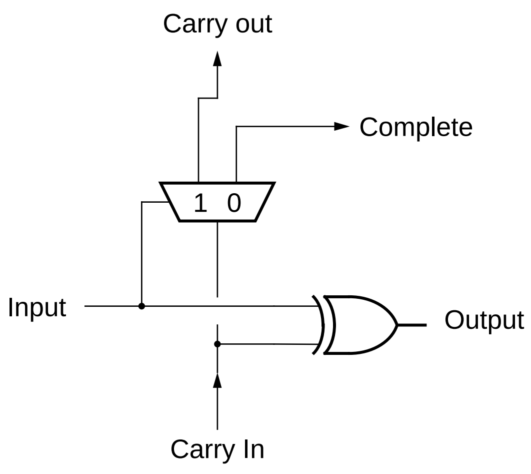


Figure 6-8: Simple ripple-carry incrementer cell

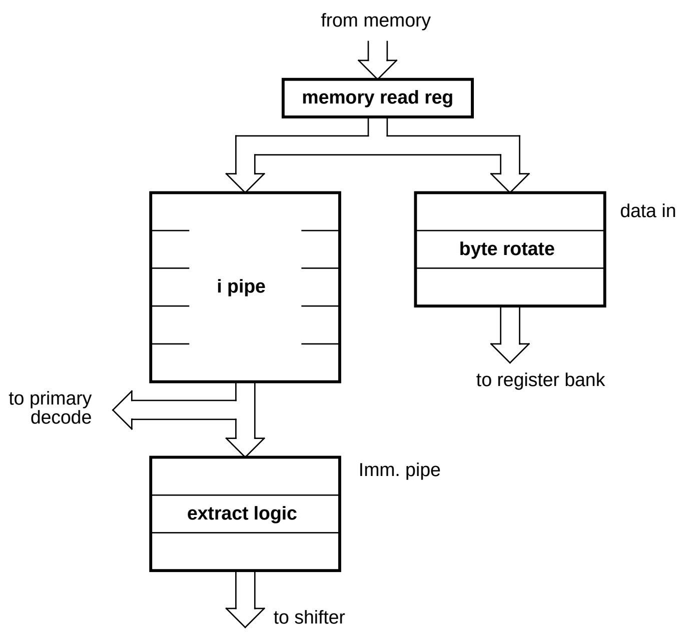


Figure 6-9: Data-in organization

carry inputs of the higher order bits do not decay after the incrementing has completed and hence cause power dissipation.

6.2 Data interface

The data interface is much simpler than the address interface. The *data out* section is simply a two stage FIFO with optional byte replication built into the input of the second stage. The *data in* logic is more complex.

6.2.1 Data in

Values arriving from memory are initially stored in the memory read register (see figure 6-9). For each value, the data interface extracts the corresponding control information from the memory control pipe, this determines whether the value is an instruction or data value.

The instruction pipeline (*Ipipe*) is a simple 5 stage Micropipeline configured as fast forward latches (see figure 3-26) to reduce latency through the *Ipipe*. The end of the instruction FIFO is connected to the primary decode and the immediate field extractor (*Imm. pipe*).

The immediate extractor consists of two Micropipeline stages with intermediate extraction logic. It is possible to determine the size of immediate value to extract (8, 12 or 24 bits) by examining just two bits of the instruction word, so the extraction logic is quite simple. The input stage is configured to be a normally closed (blocking) latch so that, when the immediate extractor is not in use, transitions on its input caused by passing instructions do not cause internal power to be dissipated. The output of the immediate extractor is connected to a multiplexer on the input of the shifter.

The final part of the data interface is the logic which processes data destined for the register bank (*data in*). This consists of two Micropipeline stages with logic to perform rotates in byte quantities for non word-aligned loads and gating to mask out the top 24 bits for byte reads. It also contains logic that can be used to alter which byte in the word is addressed as byte zero (i.e. it can change from “little-endian” to “big-endian” operation).

As the majority of values returning from memory are instructions, the data processing part has a blocking latch on its input to prevent internal power dissipation when it is not needed (i.e. when the incoming value is an instruction).

Chapter 7 : Execution pipeline

The execution unit contains the major data processing logic of the processor. It provides an autonomous multiplier, a barrel shifter connected to one of the operand buses, an ALU and storage for the Current Processor Status Register (CPSR). The interconnection of these elements is shown below in figure 7-1 (the shaded boxes represent pipeline latches).

When the multiplier is not required, there is an internal bypass mechanism which passes its inputs straight through to the outputs.

7.1 The multiplier

The multiplier in AMULET1 is substantially different from the ‘2-bits at a time’ Booth’s multiplier used in the ARM6 [Furb89, Page 253]. A multiplication using the ARM6 involves a complete cycle around the datapath for every 2 bits of the multiplicand with the intermediate results being stored temporarily in the register bank.

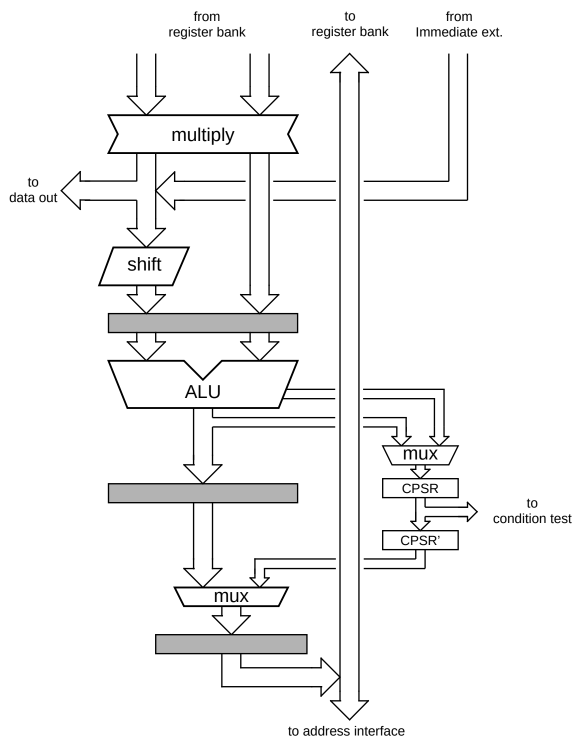


Figure 7-1: Execute pipeline

Multiplication in AMULET1 is based upon a shift-and-add multiplier using carry-save adders [Day92]. It is an autonomous unit which accepts two source operands and produces the partial product and carries which are added together in the ALU to complete the operation. Internally the multiplier controls the cycles of the shift and add operations and can terminate early when the multiplication is complete.

Each cycle of the multiplication involves:

1. Latching the inputs to the adders.
2. Performing the shift and add operation.
3. Latching the output of the adders

This continues until the multiplication is complete. The speed of operation of the circuit is governed by the speed at which the two-phase control circuitry can open and close the input and output latches. Early designs indicated that the control cycle time was much greater than that required to perform a single 1-bit shift and add operation. Investigations revealed that it was possible to do a 3-bit shift and add operation in the control cycle time so although the cycle time of the multiplier could not be improved, the amount of work done per cycle was increased and the overall performance of the multiplier improved.

As the AMULET1 multiplier is an autonomous self contained unit, it does not activate the entire datapath for every cycle of its operation (thus saving power).

7.2 The shifter

The shifter in AMULET1 is the ARM6 barrel shifter with an added matched path for self-timing purposes. The barrel shifter is constructed as a 32 x 32 cross switch matrix of n-type pass transistors. The circuit is dynamic in operation with the outputs being precharged High before a shift is evaluated. As specified by the ARM architecture, the shifter is only connected to one operand bus and is in series with the ALU.

7.3 The ALU

ARM ALU operations fall into one of three main categories: moving data from input to output, performing logic functions (XOR, AND, OR) and arithmetic operations (addition). Input buffers allow optional zeroing or complementing of operands to support subtraction (using twos-complement).

In addition to the use of the ALU for the sixteen explicit data processing instructions specified by the ARM architecture, the ALU is used implicitly by the other instruction classes (see section 4.3), for example, to calculate the address of a data transfer given the base address and an offset.

7.3.1 Performance considerations

In a synchronous system, the overall performance of the ALU is usually limited by the arithmetic logic. The speed of the addition operation is related to how quickly the carry signals can propagate across the word. The worst case occurs when the carry propagates across all bits in the word. In a synchronous system the clock period is chosen to allow time for this worst case operation although typically the result will be ready much sooner. To reduce the time for the worst case (and hence reduce the clock period) synchronous systems use schemes such as carry look-ahead or carry select [Eshr89].

Without the constraint of an external clock, an asynchronous ALU can be designed to be quick for “typical” operands and slower for worse case operands; there is no need to ensure that the worst case is equally fast, if it does not happen very often. The emphasis, with asynchronous logic design techniques, is to make the average case operation fast but to accommodate the worst case by allowing more time for the calculation. Synchronous systems cannot allow more time for the worst case because the clock period is fixed.

Optimizing the ALU design for the fast addition of “typical” operands only has benefits if the worst case is statistically rare and much worse than the average. Figure 7-2 (a) shows the mean carry propagation distance as a function of the width of the word when the data are random (figures 7-2(a) & (b) are reproduced with the kind permission of Dr. J.D. Garside [Gars93]). It can be noted that the mean carry propagation distance for a 32-bit addition is only 4.4 bits. This is much less than the worst case (32-bit).

In practice, data are not random, so to obtain a more accurate reflection of the mean carry distance a carry length analysis of a dynamic instruction trace was undertaken. The distribution of carry propagation lengths while running a benchmark program is shown in figure 7-2(b). The statistics are divided into address calculations and data operations where the average propagation distance for data operations was found to be approximately 18 bits, whereas the propagation distance for address calculations was found to be only 9 bits. Overall this gave a combined average carry propagation distance of approximately 12-13 bits. This is described in more detail elsewhere [Gars93].

The net result of the small average propagation distance during typical ALU operation is that a simple ripple-carry design can be constructed which, on average, performs better than more elaborate carry lookahead/select adders; but if the (rare) worst case is encountered it takes longer to calculate the answer.

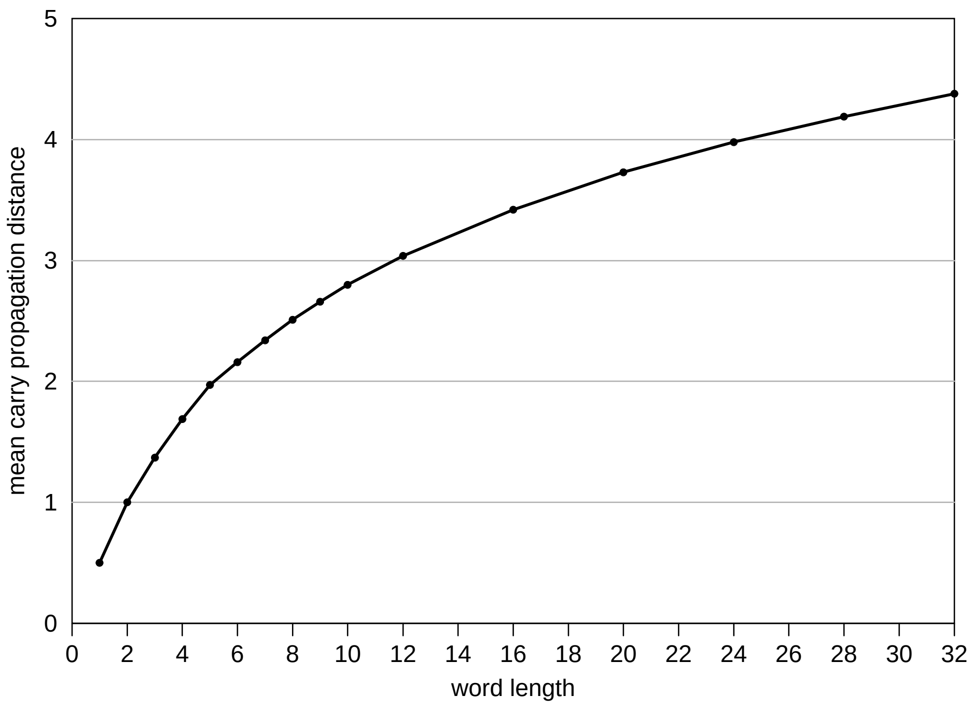
7.3.2 Implementation

The ALU adder in AMULET1 consists of thirty two full adders with no special acceleration logic to speed up carry propagation. The carry signal is encoded in dual-rail format and a completion detection circuit signals to the environment when the carry propagation is complete. The detail transistor circuitry to achieve this is described elsewhere [Gars93].

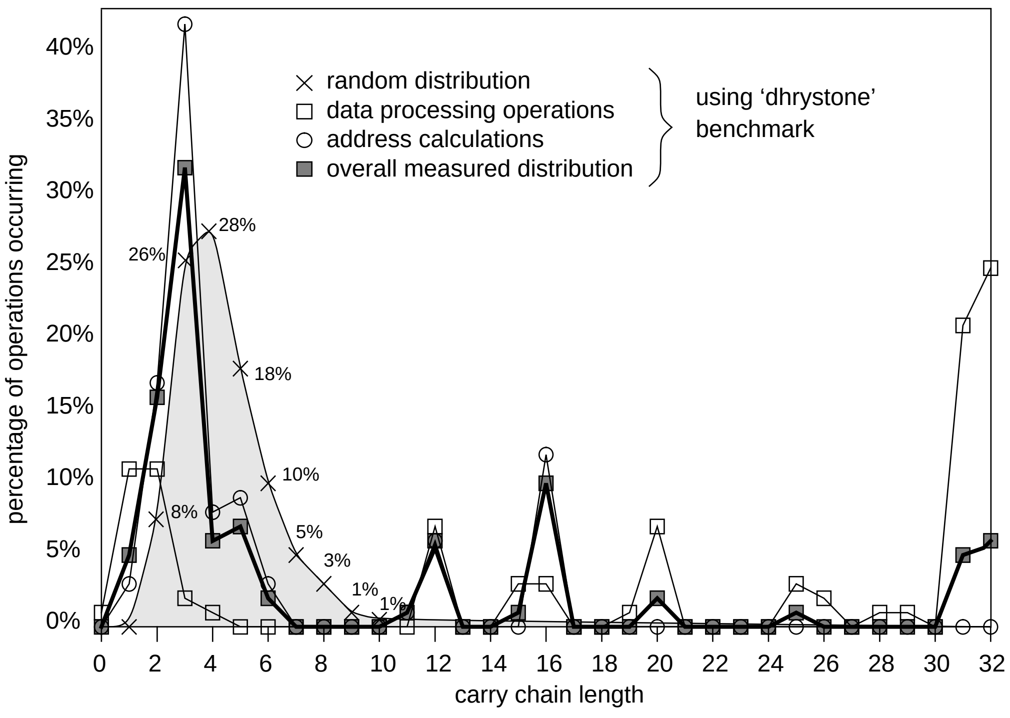
Figure 7-3 shows the resulting silicon layout of the ALU in AMULET1 in comparison to the ALU of the ARM6 drawn at the same scale. The asynchronous ALU of AMULET1 is approximately 40% of the area of its synchronous counterpart. The sparse, area-inefficient, carry-select logic can clearly be seen to the left of the ARM6 ALU.

7.4 The CPSR

The placement of the CPSR was one of the most taxing decisions taken during the design of AMULET1, it contains the arithmetic flags and the processor mode information. The arithmetic flags are generated and used in the ALU but are also made available to the condition evaluation hardware. The control flags are changed either as a result of decode information or by being directly loaded (using an ARM *MRS/MSR* instruction). The mode



(a) Average size of longest carry propagation for random data distribution



(b) Longest carry chain length distribution - 32-bit word

Figure 7-2: ALU carry propagation statistics

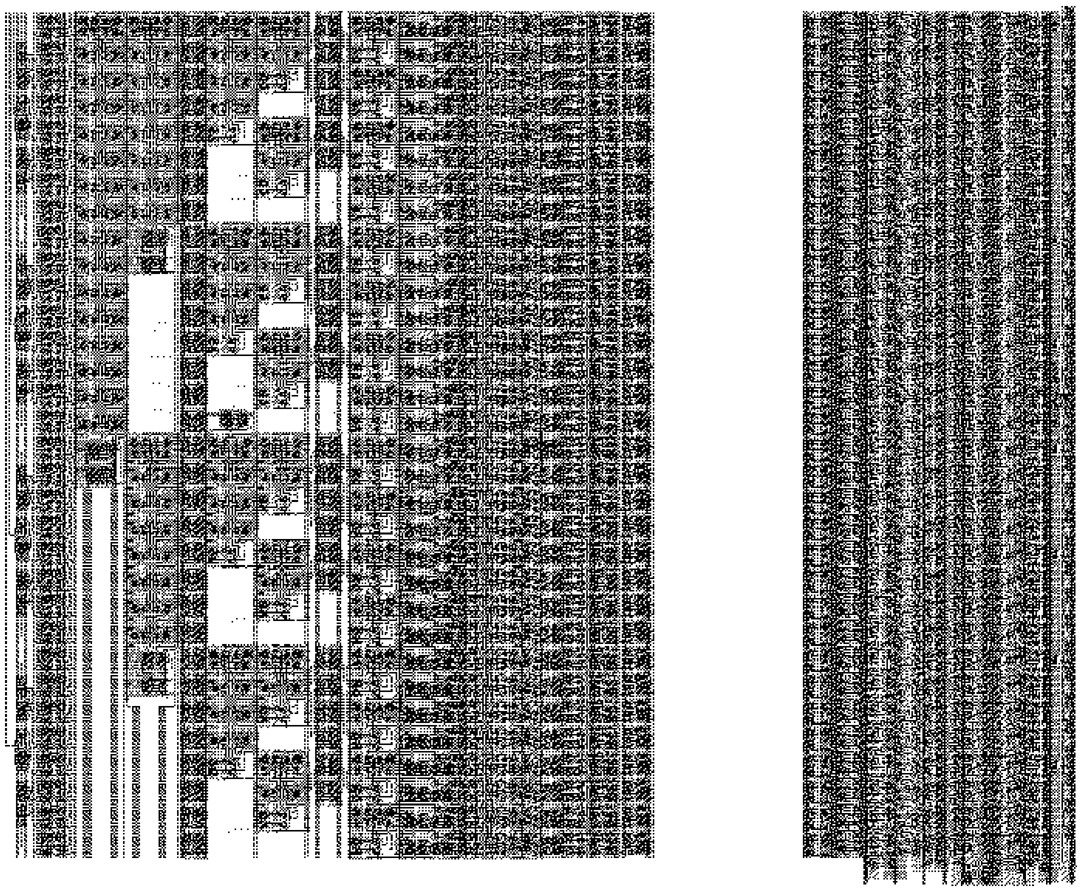


Figure 7-3: ARM6 v AMULET1 ALU area

information is used in the primary decode to determine which subset of the register bank is currently available to the programmer and in the memory interface to ensure that the correct memory privileges are enforced.

The ALU decode is the earliest point at which it is known whether an instruction will execute (i.e. whether it has passed its condition test). It is also the earliest point at which the processor state can be changed, so it is a convenient place to store the processor state. Placing the CPSR close to the ALU is also advantageous because of the close connection required between the CPSR arithmetic flags and the ALU.

An instruction which changes the CPSR mode flags also changes the visible register set (see figure B-4 for the register bank organization). The mode information is directly connected from the CPSR to the register decode circuitry, so that the correct register selection is performed. To ensure correct operation, an instruction which follows a mode changing operation in the pipeline is forced to wait at the primary decode until the potential mode change has taken place. The ALU decode (decode 3) signals to the primary decode when it is safe to continue (i.e. the mode flags are stable and will select the correct register subset).

The CPSR structure consists of 2 latches connected as shown below in figure 7-4. The active CPSR is stored in latch *A* with the output connected to the condition evaluation hardware and the relevant register decode logic. The second latch holds a copy of the CPSR while new values are being calculated. In particular, it holds the previous arithmetic flag inputs to the ALU stable while the new flags are being calculated.

The second latch also has an important role in preserving the original CPSR during exception entry. There are three cycles associated with exception entry; generate and issue the exception vector, copy the CPSR to the SPSR (of the mode being entered) and finally copy the PC into the link register of the destination mode (see section 4.3.7).

During the first cycle, the mode of the processor is changed in latch *A* at the same time as the exception vector is sent to the address interface. The original CPSR is still preserved in latch *B*. The second cycle of exception entry waits in the primary decode for the mode change to take place and then, after locking the SPSR in the new mode, it progresses to the ALU stage where the original CPSR is copied from latch *B* and sent to the SPSR of the new mode in the register bank.

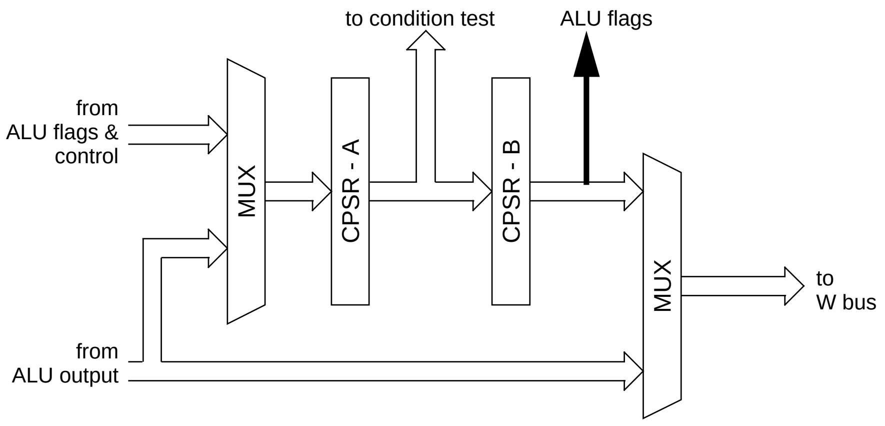


Figure 7-4: CPSR structure

Chapter 8 : Implementation

To implement a complex circuit, such as the asynchronous ARM, an ordered approach to design and verification is required. The design flow used during the construction of AMULET1 has many similarities with that used by ARM Ltd. in the design of the synchronous ARM processors [Furb89, Page 285].

8.1 Design flow

The overall design flow for AMULET1 is shown in figure 8-1. The design process begins by constructing a high-level model. Initially, *Mainsail* [XIDA87, Comp91b], the language used in the Compass Design Automation tools, was used but proved to be inefficient in this application and the resulting simulations took too long to run. Instead, ASIM [Smit92], an internal ARM Ltd. tool was used to model the processor. ASIM provides an efficient event driven simulator and a hardware description language with a library of standard parts. Extra models written in C [Kern88] can be added to the standard library.

A complete design of the processor was developed within the ASIM environment. Verification was performed by connecting the asynchronous processor to a simple simulated memory system and then loading and running the ARM validation programs. In total, over 4 million instruction cycles were simulated and it was during this time that the deadlock situations described earlier were detected (e.g. see chapter 6).

Once a stable processor design was available, this was transferred into schematic form in the Compass Design Automation environment. With detail schematics complete, it is possible to begin designing the silicon geometry. The silicon layout uses efficient custom designed cells for the datapath and standard cells from a cell library for the majority of the control logic. When the layout of a block is complete, it is compared to the schematic using automatic comparison tools (net compare [Comp91c]) to ensure it is correct.

8.1.1 Verification of the design

It is possible to generate static test vectors for a block which adheres to a Micropipeline interface. This is done by placing large delays in all event wires which cross the test interface in the high-level model. The output events of a block indicate that the output data is valid, so a large delay in the request path means that the output data is stable for longer and can be extracted as a test vector. A similar argument can be applied to input bundles with large delays in the event path.

ASIM allows any level of hierarchy to be defined as a test interface for vector generation. This allows the complete processor core to be simulated and the vectors to be generated just for the required interface (note that all event wires crossing the interface must have had the

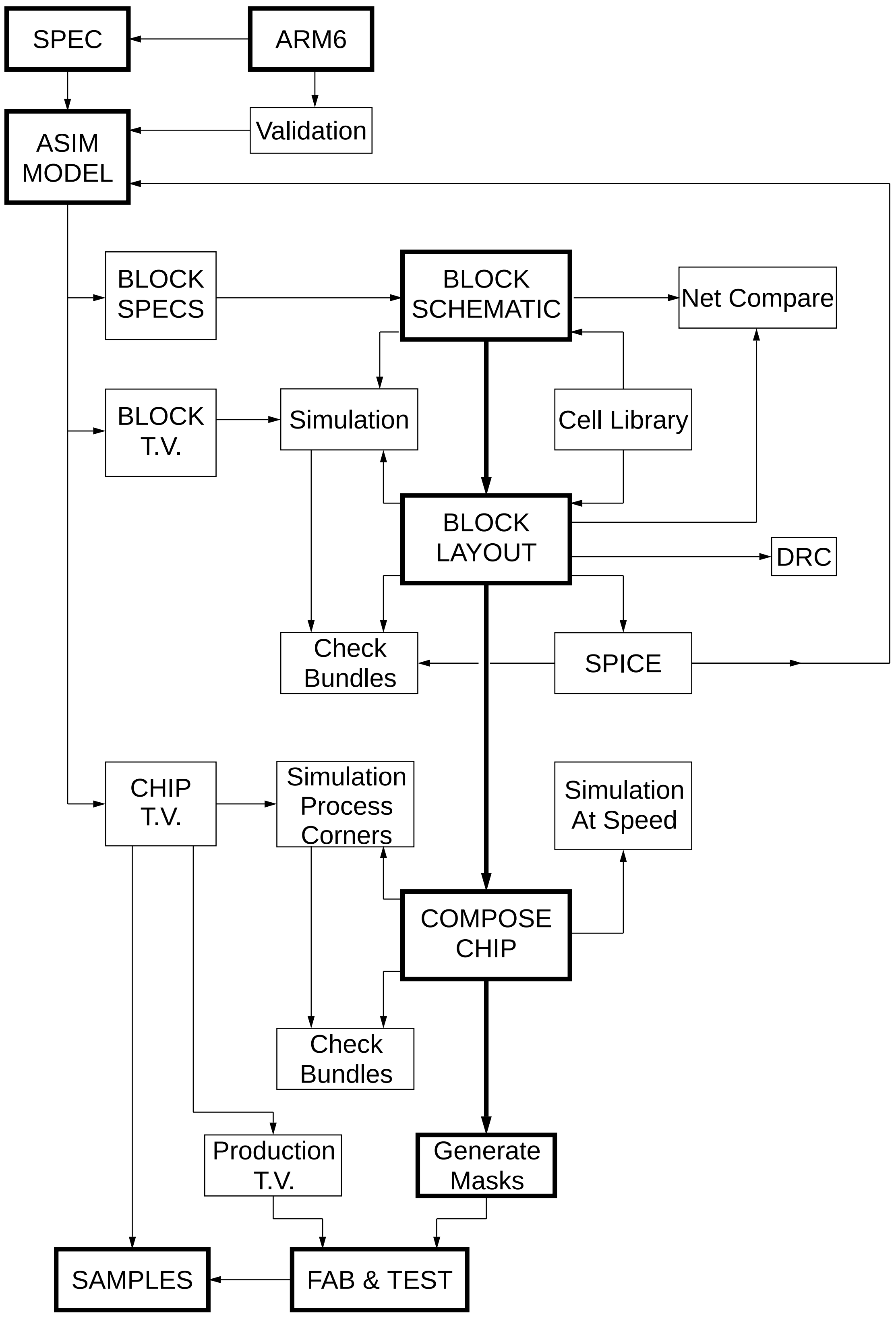


Figure 8-1: Design flow

delays inserted first). Once the vectors have been extracted, ASIM provides a facility to verify that the vectors are static by re-applying them directly to the block to be tested and observing the outputs. Experience has shown that it is relatively straightforward to generate static vectors for a clean Micropipeline interface.

The block level test vectors are translated into the Compass Design Automation environment and can be used to verify the block level schematics and subsequently the extracted silicon layout.

An important consideration of a Micropipeline system is to ensure that all interfaces obey the bundle data convention. This constrains the output data to be valid before an output request is issued and the data must remain stable until an acknowledgement is received (see figure 3-2 in chapter 3).

The bundle constraints are checked at two levels, first by using SPICE to examine any matched paths or self-timing and then, at a higher level, a simple tool was written to check the simulator output. Given a definition of data bundles and their corresponding control, it is relatively straightforward to check that the data was stable before a transition on the request wire and that it did not change until after a transition on the acknowledge wire.

The same verification approach was applied to the complete chip after the blocks of layout were composed and the top level wiring connected. In addition, all test vector simulations were run at all four “process corners” of the silicon (i.e. all combinations of fast and slow n- and p-transistors). This can give a effective speed ratio of n- to p-transistors of 4:1 in both directions.

To allow the design to be verified “at speed” (rather than using just static vectors), the extracted layout of the complete chip was connected to a simulated system. The memory of the system was then loaded with a program and the processor allowed to execute it at its own speed. This “at speed” testing applies only to the particular point in the possible process space to which the simulator is calibrated; to increase test confidence, the program was executed at all four simulated process corners (again using a 4:1 spread in transistor speeds).

The complete chip has a Micropipeline interface, so static test vectors can be generated as described before. Programs carefully chosen to toggle a high percentage of internal nodes can be used to generate the static production test vectors required to verify the chip after fabrication. The static nature of the vectors allows conventional testers to be used.

Much work still needs to be done to address the problem of test and verification of an asynchronous processor (fault simulation etc.). As many parts of the asynchronous datapath are essentially the same as their synchronous counterparts, some of the synchronous test techniques may be applied to them. However, techniques to verify the control need to be further developed.

8.2 Complete organization

The overall complexity of the resulting design is shown in figure 8-2. The diagram features details not previously illustrated:

- The memory control FIFO (*mem ctrl. FIFO*) is connected from the address control information latch to the destination control (*dest. ctrl.*) in the data interface to

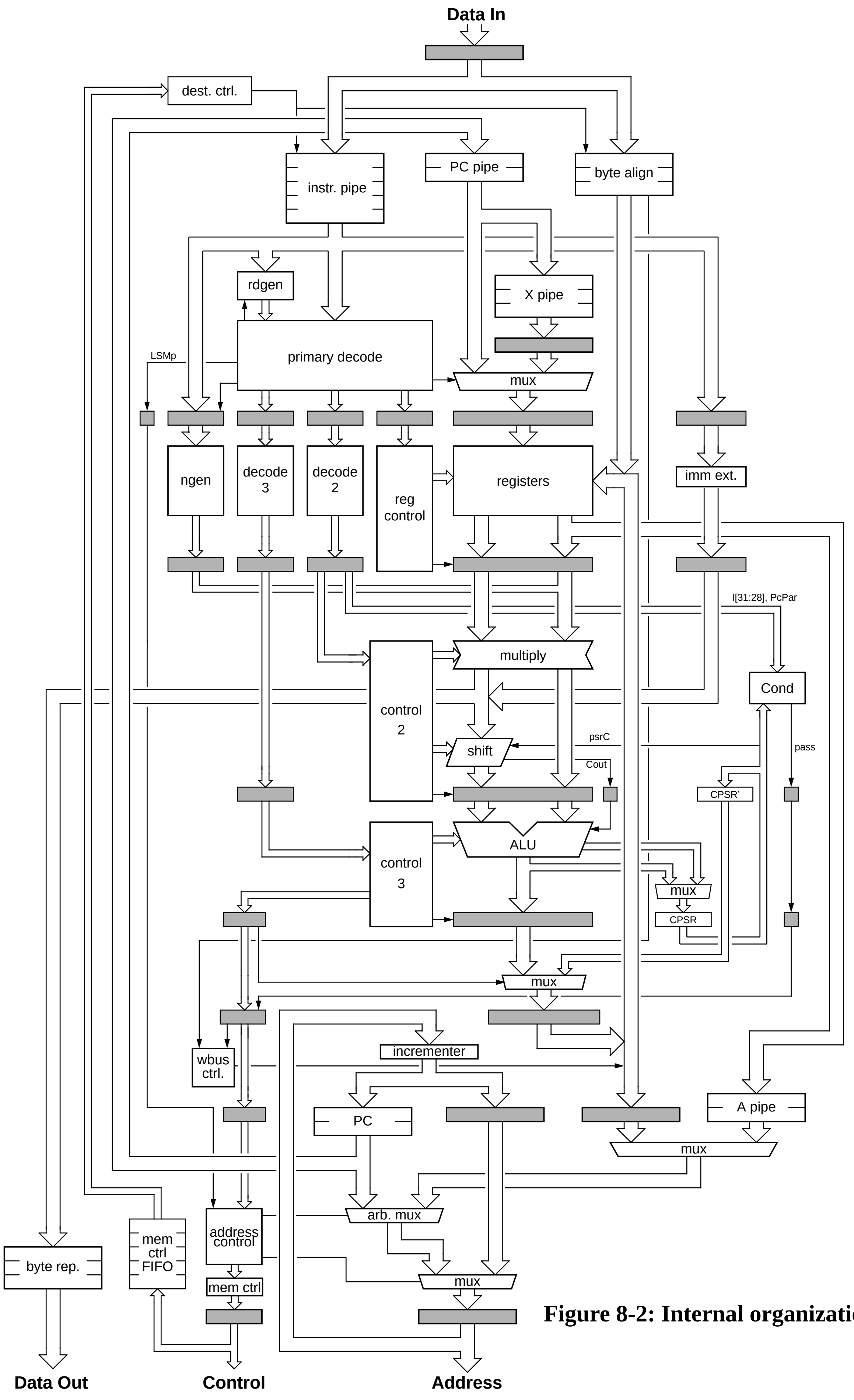


Figure 8-2: Internal organization

allow the incoming memory values to be steered to the correct destination (either the instruction pipeline (*instr. pipe*) or the data-in processing logic (*byte align*)).

- The primary decode and the address interface are connected via the *LSMp* pipeline to facilitate the load/store multiple instruction (see section 4.3.6 & section 6.1.1).
- *Rdgen* which sequentially generates the addresses of the registers to transfer in a load/store multiple instruction given the sixteen bit field from the instruction word (1 bit is set for each register to transfer).
- *Ngen* calculates the number of registers to be transferred in a load/store multiple instruction from the bottom sixteen bits of the instruction. The resulting value is used in base calculations (see section 4.3.6). It is also convenient for *ngen* to generate the vector addresses for exception entry (see section 4.3.7) because of its connection to the datapath *A* bus.
- The write bus control logic (*wbus ctrl.*) which arbitrates requests from either the ALU result register or incoming memory data wishing to use the write bus.

8.2.1 Datapath VLSI organization

Figure 8-3 shows an implementation orientated view of the overall organization, with particular emphasis on the VLSI floorplan of the datapath. The major data buses are shown in blue and the major control dependencies are shown in red. The actual order of the datapath blocks in the diagram are as implemented in silicon. The diagram also shows how some of the multiplexers, shown in figure 8-2, are implemented as a shared bus with each possible source having tri-state drivers onto the bus. For example, the ALU output latch and the CPSR in the previous diagram are shown as being combined with a multiplexer into the result latch. The actual implementation shows a shared bus arrangement.

8.3 Silicon layout

The complete silicon layout of AMULET1 is shown in figure 8-4. This is annotated with the major block names so that it can be related to earlier figures in this chapter. The lower half is the regular, custom designed, datapath, the order of which is detailed in figure 8-3. The control section is predominately compiled standard cells. The cells are taken from the basic cell library from ARM Ltd. with the addition of the Sutherland elements described in chapter 3. There are two areas of the control that are implemented as PLA structures. These are synthesized using a modified version of the ARM Ltd. PLA generator [Howa89].

Internally, the PLAs are dynamic with a self-timed matched path completion signal which powers down the PLA after the results are latched. The modifications for use in AMULET1 simply made the completion signal available at the PLA interface. The PLA circuit, as used in the ARM6, is a good example of a high volume self-timed circuit in use today and shows that it is possible to have reliable self-timed circuits.

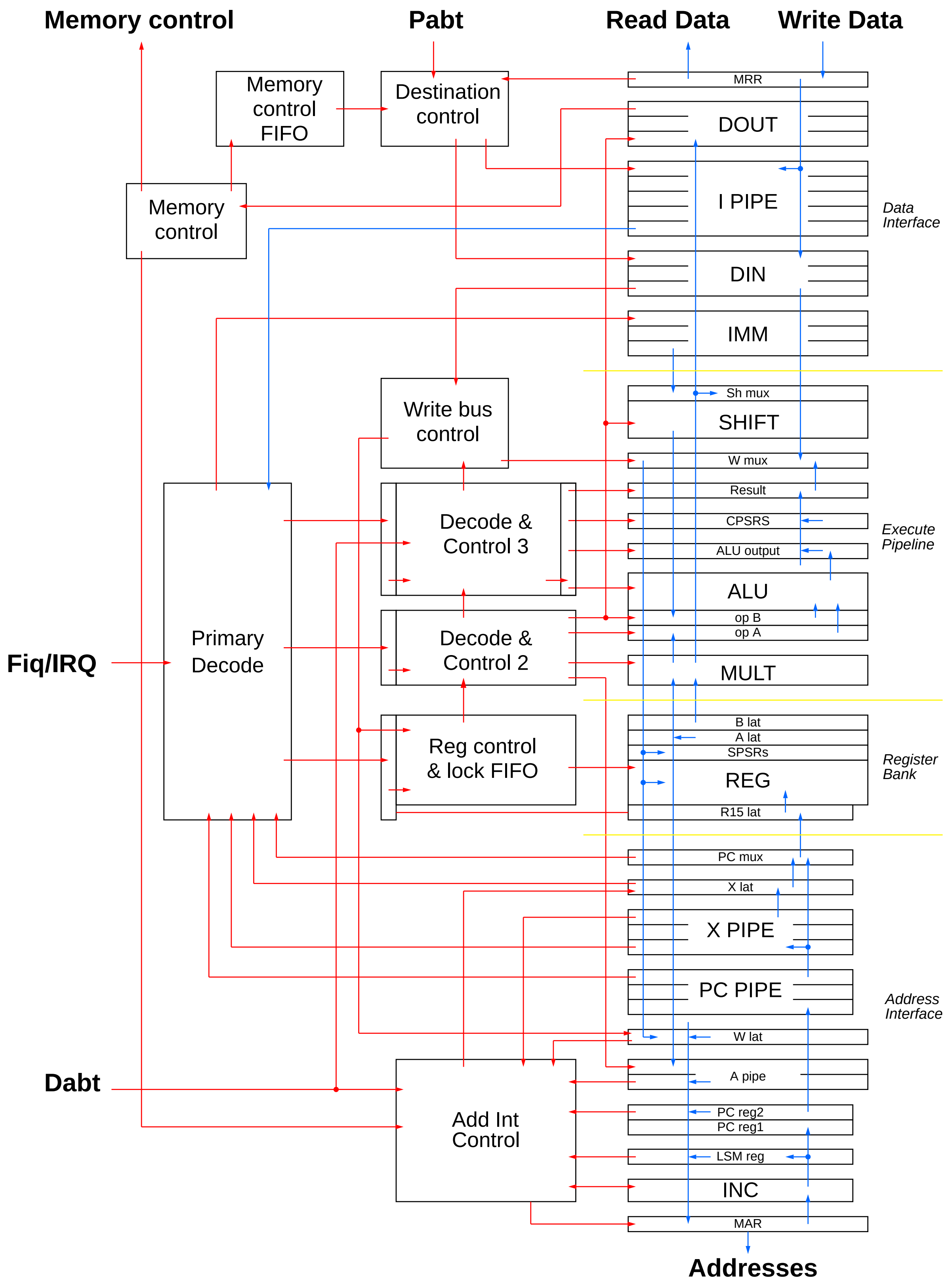


Figure 8-3: Organization of the datapath and control dependencies

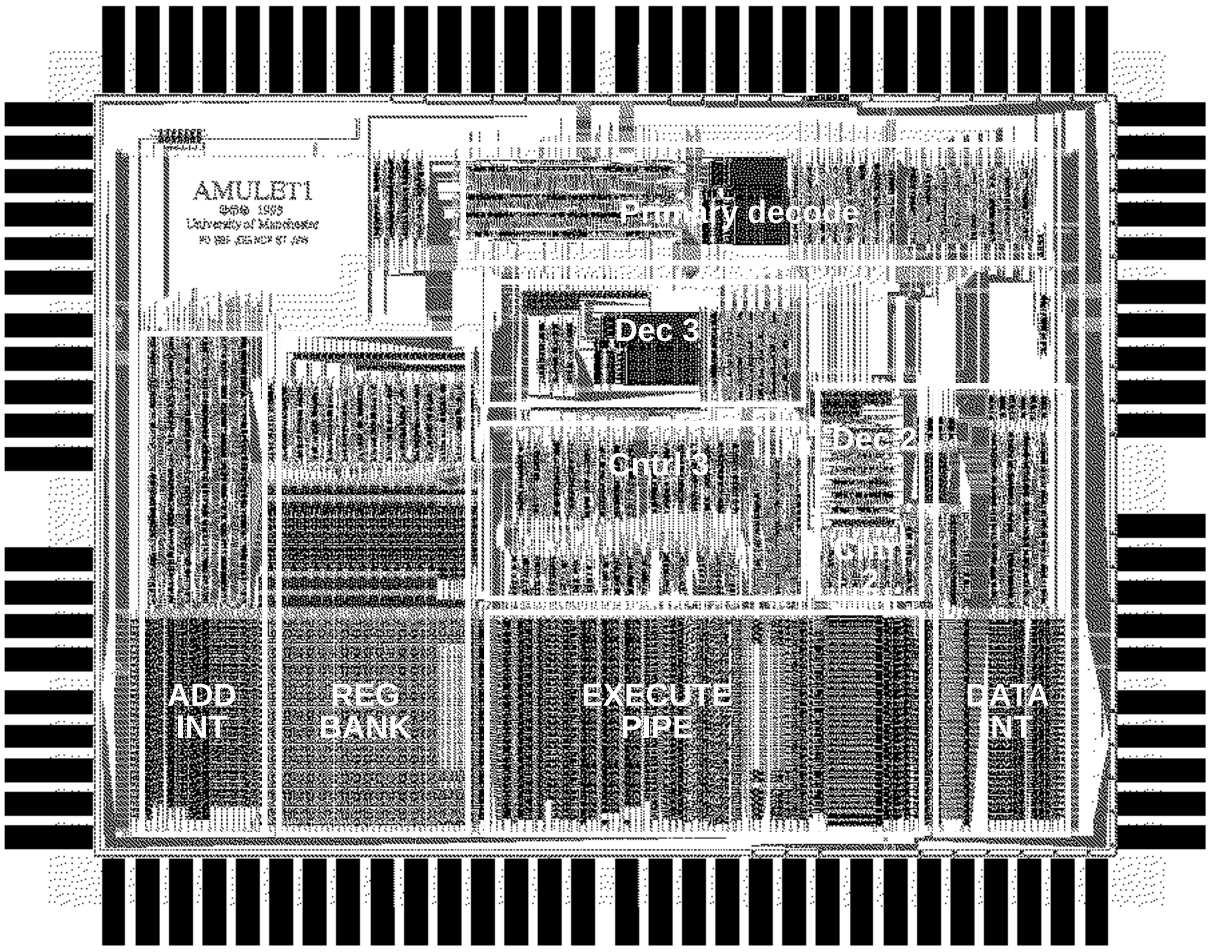


Figure 8-4: AMULET1 1.2 micron physical layout

8.4 Test devices

Although the design of AMULET1 was targeted at the VLSI Technologies CMN12 (1.2 micron) process, the first test devices were fabricated on a 0.7 micron CMOS process by GEC Plessey Semiconductors. The silicon layout was translated onto the new process by a series of semi-automatic geometry transformations applied by the foundry.

AMULET1 was fabricated as part of a multi-project wafer. It shares a single gate array pad ring with other third party test circuits, with only one of the test cores connected to the pad ring in any particular die (the last metal layer determines which core is connected to the pad ring). Figure 8-5 shows a plot (from CAD tools) of the organization of the die with the AMULET1 core clearly visible in the lower left corner.

Figure 8-6 shows a photograph of the fabricated devices mounted in a PGA package. The organization of the die is as in shown the previous figure (figure 8-5) with the AMULET1 core in the lower left corner. All 256 pads are bonded out to the package but the AMULET1 core is connected to only 100 pads.

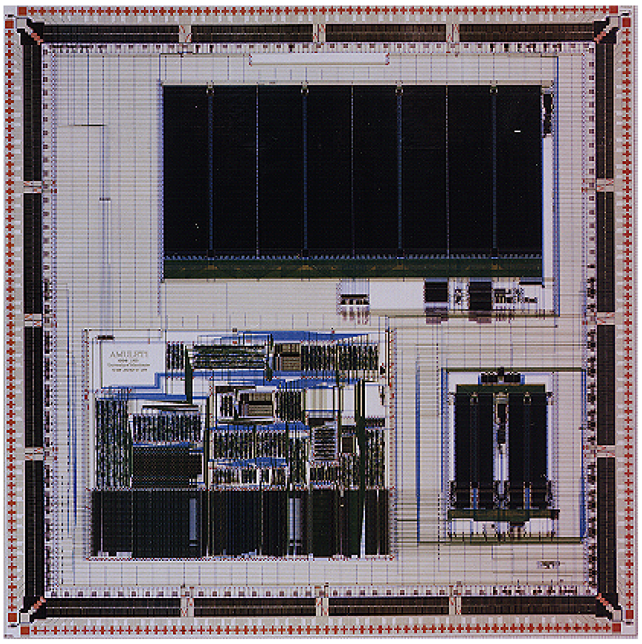


Figure 8-5: AMULET1: 0.7 multi-project die organization

Figure 8-6: AMULET1: 0.7 micron multi-project die

Chapter 9 : Evaluation and further work

Twenty four (0.7 micron) test devices were delivered from the foundry; ten were delivered untested and fourteen had been successfully wafer probe tested (using the static test vectors that were generated from ASIM - see section 8.1.1).

A printed circuit board test card was constructed to evaluate the devices further. The board consists of an AMULET1 device, 128K of RAM, 128K ROM, a UART (for a serial interface) and associated control logic. The test card was designed to be compatible with the standard ARM6 PIE card [ARM92] so that the ARM6 debug monitor[1] (contained in a ROM) could be used to analyse the AMULET1 design.

The debug monitor ran successfully on the AMULET1 test card and enabled the card to communicate with a host machine to download and run programs. All the ARM validation programs which could run on the test card completed successfully[2]. Of the ten untested devices seven were found to be functional, and of the fourteen probe tested, twelve were found to be functional after packaging

Preliminary performance measurements using the Dhrystone benchmark have shown AMULET1 operating at 28K Dhrystones; further refinements to the test card could well improve on this. With no on-chip cache the performance of AMULET1 is likely to be limited by the speed of the test card memory system.

The power consumption of the AMULET1 core cannot be measured separately on the first test devices because the core and pads share the same power supplies (this is a consequence of the multi-project die). Further devices are being fabricated (on a CMN12 compatible process at a different foundry) with the core power supplies separated to enable accurate power measurements to be taken.

9.1 Design characteristics

As a basis for comparison between AMULET1 and ARM6, table 9-1 shows the characteristics of both chips on the CMN12 process, using performance and power figures obtained from simulations under *slow-slow* conditions.

1. ARM60-PIE DEMON V1.0, ARM Ltd., 1991.
2. Some of validation programs are written in such a way that they must be loaded at address location 0 in memory and therefore cannot run under the debug monitor because it also uses the same memory area.

Table 9-1: Characteristics of the AMULET1 compared with ARM6

	AMULET1	ARM6
Process	1.2 μm DLM CMOS[a]	1.2 μm DLM CMOS
Cell core area	5.5mm x 4.1mm	4.1mm x 2.7mm
No. of transistors	58,374	33,494
Performance	9 K Dhrystones[b]	14K Dhrystones @ 10MHz
Dissipation	83mW[c]	75mW @ 10MHz
Design Effort (approx.)	5 man years	5 man years

a. Double Layer Metal CMOS

b. Simulated performance under slow-slow conditions

c. Estimated power consumption from simulation

This table shows that the silicon implementation of AMULET1 is approximately twice the cell area of an ARM6 on an equivalent process. This is also reflected in the 75% more transistors used in AMULET1 compared to ARM6.

The performance figures for AMULET1 shown in table 9-1 are based upon worst-case[1] simulation of the device running Dhrystone code compiled using a standard ARM compiler. The ARM6 figures are the worst-case specification from the data sheet[2] [ARM91b]. The difference between the estimated performance (shown in table 9-1) and that achieved by the test devices can be accounted for by observing that the supplied devices represent *typical* silicon[3] which is usually twice the performance of *slow-slow*, and the transition from a 1.2 micron process to a 0.7 micron process results in approximately 30-50% speed improvement (this figure is hard to quantify exactly).

The power dissipation figures for AMULET1 are taken from a tool which monitors all extracted layout nodes in a simulation and calculates the energy dissipated whenever there is a transition at a node [Davi94]. The power is calculated by summing the energy dissipation at all nodes and averaging this value over time. The results from this power analysis tool have yet to be verified but this is the best estimate for power consumption at present.

The design effort required to design and implement the asynchronous organisation of AMULET1 was comparable to the cost of producing its synchronous counterpart.

9.1.1 Area overhead

The organization of AMULET1 employs a relatively deep pipeline, which accounts for much of the increase in transistor count and die size relative to the ARM6. The ARM6 also

1. Worst-case simulation is when both n- and p-transistors are characterised as slow and the process is known as *Slow-Slow*.
2. Production ARM6 devices have been qualified at significantly higher clock rates.
3. GEC Plessey Semiconductors have measured the silicon and classified it as *typical*.

has more compact silicon layout (approximately 14% of the AMULET1 die area is under utilised).

The area overhead of the asynchronous control of AMULET1 can be estimated by considering a synchronous implementation of a similar pipeline organization. The datapath part of the design would remain broadly similar but the control for the latches etc. would be driven from the global clock rather than individual control circuits communicating between themselves. The control area overhead can therefore be derived by calculating the area consumed by these individual control circuits and their event communication paths. A first order approximation of this cost can be given by assuming all event control modules form the overhead. Table 9-2 shows a breakdown of the event control module use in AMULET1 and the percentage of the utilised core area they use.

Table 9-2: AMULET1 event control module area overhead

Module	Number	% of Total Core Area
XOR	139	0.94
CGate	98	0.86
TOGGLE	67	2.9
SELECT	56	2.15
CALL	13	0.52
ARBITER	5	0.28
DWAIT	7	0.22
Total module area		7.87
Wiring cost		7.87
Total area overhead		15.74

As well as the area of the modules themselves there is also an overhead associated with their wiring interconnect. By examining the existing standard cell areas in AMULET1 it can be noted that each row of standard cells has, on average, a similar area of wiring interconnected associated with it. Therefore the estimated wiring cost given in table 9-2 is the same as the module area. The table shows the total estimated cost of the asynchronous control to be approximately 16% of the total core area of AMULET1.

9.1.2 Pipeline organization

Normally it would be expected that a deeper pipeline would result in an increased throughput, however in AMULET1 this is not the case because of several factors:

- The primary decode takes too long to process instructions, so the pipeline is never busy.
- Register dependencies between consecutive instructions cause the pipeline to stall.

- The depth of the pipeline increases the cost of branches.

Time and resource pressure in this first asynchronous design did not allow for the primary decode to be modified once it was realised that it represented a bottleneck, but this can be addressed in the future. The problem of register dependencies is a function of the way the compiler allocates registers and schedules instructions. The standard ARM compilers take no account of inter-instruction dependencies because the performance of the ARM6 is not affected by them, however the performance of AMULET1 can be adversely affected by inter-instruction dependencies. To achieve optimum performance the compiler must therefore schedule instructions to avoid dependencies between consecutive instructions where possible. Preliminary investigations have shown that it is possible to improve the performance of AMULET1 significantly by including such compiler optimizations.

The cost of inter-instruction dependencies remaining after these compiler optimizations have been applied can be reduced by modifications to the processor organization to include the last result re-use and register through-passing techniques described in chapter 5.

Pipeline depth

Retrospective analysis of the AMULET1 design has revealed that the depth of pipelining is too great. This is partly due to FIFO buffers being conceptually easy to use within the Micropipeline design style, and as a result too many were added. There are many stages that contribute little (or nothing) towards performance but still cost silicon area, transistors and power dissipation (and some stages actually decrease performance!).

The depth of the pipeline also has an adverse effect on the branch latency[1]. As branches represent approximately 20% of ARM instructions [Furb89] (i.e 1 in 5 instructions) the increased latency significantly affects performance.

9.2 Further work

Further work (sponsored by the ESPRIT OMI-DE project) is already under way in several areas. The main objectives of the work are to revise the asynchronous core to reflect the experience gained during the first implementation and to add an on-chip cache and MMU. The aim of the core modifications are to improve its performance although there is much work to do before an exact figure for the extent of this improvement can be given. Some of the proposed modifications are discussed here and are split into four sections: base technology, processor organisation, development tools and test.

9.2.1 Base technology

In chapter 3 we saw that there were trade-offs in different latch styles in terms of area and speed. Preliminary investigations have show that is possible to improve performance and reduce power consumption by adopting a Svensson [Sven89] style latch, as shown in figure 9-1. This latch requires only a single enable signal (c.f. transmission gate latches which require the true and complement of the enable signal) so simplifying the control circuitry required. The enable transistors can be much reduced in size in comparison to the standard

1. The branch latency is defined to be the time from a branch beginning execution until the time an instruction from the branch target begins execution.

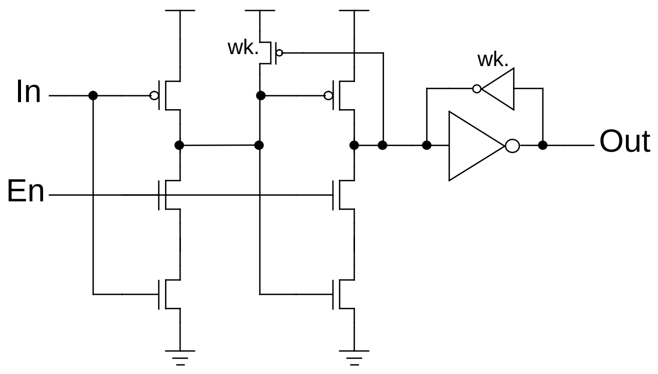


Figure 9-1: Svensson style latch

latch and so the capacitive load on the enable line is reduced. Both of these factors reduce power consumption and improve performance. This work is being undertaken in conjunction with the OMI-HORN project [Day93].

Investigations are also under way to determine the cost of two-phase signalling in relation to four-phase. To achieve this, the primary decode is being re-implemented in four-phase logic so that a realistic comparison can be made.

9.2.2 Processor organization

There are several areas of the processor organization which will be revised in future versions of the AMULET processor. The technique of last result re-use and register through passing described in chapter 5 will be included in new designs to reduce the cost of inter-instruction dependencies.

The primary decode in AMULET1 is one of the major performance bottlenecks in the current design. Time pressure in the original design did not allow this to be modified but future versions will incorporate improvements in this area.

Many of the pipeline stages in AMULET1 contribute nothing to the overall performance of the design, therefore future version of the asynchronous processor will have a much reduced pipeline structure with some pipeline stages merging. (Current investigations are considering the viability of merging the shift/multiply stage with the ALU stage. Logic would be provided to bypass the shifter when it is not required or a shift of zero is specified).

Support for a co-processor interface will be added in future versions to enable an on-chip cache controller and memory management unit to act as a co-processor. There are no plans to add an external co-processor interface at present.

9.2.3 Tools

Work has begun to investigate the compiler optimization strategies which would be appropriate for the asynchronous organization of AMULET1. This involves taking an existing compiler (GCC [Stall92]) and modifying the code generating “back-end” to perform the required optimization. An example of the type of optimization being considered is to reduce the inter-instruction register dependencies of adjacent instructions by code re-ordering. If the number of register dependencies is reduced then the register bank will stall less frequently on register locks so the performance should increase accordingly.

There is a need to develop formal tools to help analyse potential deadlock situations and prevent them occurring. To this end, a high level model of the entire processor is being constructed in Occam as part of other research [Theo93]. The Occam model is a more appropriate starting point for any possible formal reasoning than the detailed transistor schematics.

To verify the integrity of the layout produced, it would be convenient to have tools which could verify statically that the bundle constraints have been satisfied. This could operate in a similar manner to a standard synchronous timing verifier [Comp91d]. Given the completion circuitry and the data circuitry it should be possible to check statically that the bundle constraints are satisfied provided there is no internal feedback. As Micropipelines operate in fundamental mode (section 1.2.2), all feedback must flow through a latch stage (and hence a Micropipeline interface). This means it is possible to break all feedback loops if the correct level of abstraction is chosen.

9.2.4 Test

One of the least explored areas that needs addressing in future work is that of test and verification. The design of AMULET1 incorporated a very ad-hoc approach to test. Static vectors were generated from the high-level model by placing large delays in the event control wires passing through the test interface. By this means it was possible to generate static vectors for the complete chip, so the resulting device could be tested by a conventional tester.

Further research has begun in conjunction with the University of Hanover to investigate the testability of asynchronous circuits (University of Hanover is a partner in OMI-DE).

Chapter 10 : Conclusions

The design of AMULET1 has achieved the objectives set out for this work by demonstrating the feasibility of implementing a complex commercial RISC architecture using asynchronous design techniques. The resulting design addresses many of the difficult issues associated with modern RISC processors such as support for exact exceptions, backwards instruction set compatibility and pipelined operation.

The AMULET1 design exhibits several innovative features:

- The incorporation of a simple ripple-carry asynchronous ALU adder with a data dependent propagation time which performs better, on average, than more elaborate carry select adders.
- A novel arbiter-free register coherency mechanism which allows register read and writes with arbitrary timing and also allows internal ALU cycles to overtake slower external memory accesses (assuming there are no register dependencies between the two).
- An instruction prefetch unit which has a non-deterministic (but bounded) prefetch depth beyond a branch.

The design also includes a pipeline structure which allows asynchronous concurrent operation of internal functional units.

10.1 Micropipelines

Micropipelines appear to offer a good engineering framework for the design of an asynchronous microprocessor and the design details have much in common with synchronous design. The re-use of synchronous library elements is a major advantage of the Micropipeline approach over other asynchronous design styles, especially for data processing operations where many bits are processed in parallel and area and efficiency are important. In AMULET1 it was possible to use the existing silicon layout of the ARM6 register cells and barrel shifter simply by adding a self-timing path to each. This yielded a very efficient, compact datapath layout with a comparable cost to the synchronous implementation. The overhead of using the asynchronous Micropipeline control has been estimated to be approximately 16%.

The flexibility of the asynchronous Micropipeline approach allows designs to be optimized for typical operating conditions and allows resources to be concentrated on functions that are used most frequently with rare worst-case functions being allowed more time to evaluate (c.f synchronous design where resources are used to speed-up the rare worst-case functions). The asynchronous design style does, however, prevent some of the architectural enhancements often associated with deeply pipelined clocked processors (e.g. register forwarding) from being used, but alternative solutions to these problems can be adopted (e.g. last result re-use). The Micropipeline design style makes FIFO buffers conceptually easy to use, however care must be taken to ensure that their ease of use does not lead to them being used too liberally (as was the case in AMULET1).

The robustness of the Micropipeline design style has been demonstrated by the fact that a circuit which was originally designed for a 1.2 micron process has been successfully translated onto a 0.7 micron process (with all process translations being performed semi-automatically at the silicon foundry). The effort required to design and implement AMULET1 using the Micropipeline methodology was comparable with that required for the synchronous processor (5 man years).

10.2 AMULET1

The asynchronous design is within a factor 2 of the synchronous ARM6 in the important parameters of performance, silicon area and power consumption, but at present it does not show any significant advantage over its clocked counterpart. However, AMULET1 is a first attempt at a Micropipelined design of this complexity and scale, whereas the ARM6 is a fourth generation synchronous processor and a world leader in its class for small die area and power-efficiency. In addition the ARM instruction set contains many features that were defined as an artifact of the original synchronous implementation. This has caused extra complexity in AMULET1 because of its extended pipeline structure[1] and its asynchronous operation.

The performance of AMULET1 is also degraded by conservative engineering margins; in the first design the emphasis was placed upon functionality rather than outright performance, and there is considerable scope for enhancing the speed and power efficiency of the design in the future.

10.3 An asynchronous future?

Asynchronous design styles are enjoying a resurgence of interest at present due to their immunity from clock skew and their potential for high performance and power efficiency. Many questions remain to be answered before this potential can be fully realised. Up to now, one of these questions has related to the feasibility of designing effective asynchronous circuits at the level of complexity required for commercial applications. AMULET1 answers this question by providing a convincing demonstration that complex asynchronous circuits are, indeed, feasible.

1. A synchronous ARM with a similar pipeline structure would also encounter some of the same problems.

Chapter 11 : Bibliography

[Akel91] Akella V., Gopalakrishnan G., Hopcp: A concurrent hardware description language. Technical Report UUCS-TR-91-021, Department of Computer Science, University of Utah, 1991.

[Asyn93] Asynchronous Online Bibliography, accessed via electronic mail addressed to ‘async-bib@win.tue.nl’.

[ARM91a] Library Cell SPICE simulations for CMN12 process. ARM Ltd., Cambridge, England, Confidential, 1991.

[ARM91b] ARM6 Macrocell datasheet. ARM Ltd, Cambridge, England, September 1991.

[ARM92] ARM PIE User Guide. ARM Ltd, Cambridge, England, 1992.

[Beer92] Beerel P.A., Meng T.H-Y., Automatic Gate-Level Synthesis of Speed-Independent Circuits. Proc. Int'l. Conf. Computer-Aided Design. IEEE Computer Society Press, November, 1992.

[Brun89] Brunvand E., Sproull R.F., Translating concurrent programs into delay-insensitive circuits. In ICCAD-89, pp.262-265, IEEE, November, 1989.

[Brun91] Brunvand E., Translating Concurrent Communicating Programs into Asynchronous Circuits. PhD thesis, Carnegie Mellon University, 1991.

[Brun91a] Brunvand E., Starkey M., An Integrated Environment for the Design and Simulation of Self Timed Systems. VLSI 91, pp. 4a.2.1-4a.3.1. IFIP, August, 1991.

[Brun91b] Brunvand E., A Cell set for self-timed design using Actel FPGAs. Technical Report UUCS-91-013, University of Utah, 1991.

[Brun91c] Brunvand E. Implementing self-timed systems with FPGAs. In FPGAs, chapter 6.2, pp. 312-323, editors W.R. Moore and W.Luk, Abingdon EE&CS Books, 1991.

[Brun92] Brunvand E., Michell, N., Smith, K. A, Comparison of Self-Timed Design using FPGA, CMOS, and GaAs Technologies. 1992 IEEE International Conference on Computer Design: VLSI in Computers & Processors, pp. 76-80. October, 1992.

[Brun93] Brunvand E., The NSR Processor. 26th Hawaii Int. Conference on System Science (HICSS 1993), pp. 428-435. January, 1993.

[Burn87] Burns S.M., Automated Compilation of Concurrent Programs into Self-timed Circuits. MSc thesis, Computer Science Department, Caltech, 1987.

[Burn88] Burns S.M., Martin A.J., Synthesis of self-timed circuits by program transformation. In The Fusion of Hardware Design and Verification, pp. 99-116, editor G.J. Milne, Elsevier Science Publishers, 1988.

[Chan73] Chaney T.J., Molnar, C.E., Anomalous Behavior of Synchronizer and Arbiter Circuits. IEEE Transactions on Computers. C-22(4):421-422, April, 1973.

[Chu85], Chu T.A., Leung C.K.C., Wanuga T.S., A Design Methodology for Concurrent VLSI systems. Proceedings of ICCD, pp 407-410, 1985.

[Chu86a] Chu T.A., On the Models for Designing VLSI Asynchronous Digital Systems". INTEGRATION the VLSI Journal , Vol. 4, pp. 99-113, 1986.

[Chu86b] Chu T.A., Glasser L.A., Synthesis of Self-timed Control Circuits from Graphs : An Example. Proceedings of ICCD, pp.565-571, 1986.

[Chu87] Chu T.A., Synthesis of Self-timed VLSI Circuits from Graph-Theoretic Specifications. PhD thesis, M.I.T., MIT/LCS/TR-393, June 1987.

[Clar67] Clark W.A., Macromodular Computer Systems. AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference, pp. 335-336. Academic Press, Atlantic City, NJ, 1967.

[Clar74] Clark W.A., Molnar C.E., Macromodular Computer Systems. Computers in Biomedical Research, editors Stacy R.W., Waxman B.D. chapter 3, pages 45-85. Academic Press, 1974.

[Coat93] Coates B., Davis A., Stevens K.S., Automatic Synthesis of Fast Compact Self-timed Control Circuits. Proceedings of the IFIP working conference on Asynchronous Design Methodologies, Manchester, England, 1993.

[Coat93a] Coates B., Davis A., Stevens K.S., The Post Office Experience: Designing a Large Asynchronous Chip. to appear in INTEGRATION, 1993.

[Cock87] Cockerell P., ARM Assembly Language Programming. MTC, England, 1987.

[Comp91] Chip Compiler V8R3. Compass Design Automation Inc., San Jose, U.S.A., Document No: VSD 40360, 1991.

[Comp91a] Layout Editor V8R3. Compass Design Automation Inc., San Jose, U.S.A., Document No: VSD 10414, 1991.

[Comp91b] Behavioral Modeling Langauage. Compass Design Automation Inc., San Jose, U.S.A., Document No: VSD 19160, 1991.

[Comp91c] Netlist Comparison. In Layout Verification, Compass Design Automation Inc., San Jose, U.S.A., Document No: VSD 50107, 1991.

[Comp91d] QTV Timing Verifier. In Verification Tools, Compass Design Automation Inc., San Jose, U.S.A., Document No: VSD 41400. 1991.

[Cour75] Couranz G.R. and Wann D.F. Theoretical and Experimental Behavior of Synchronizers Operating in the Metastable Region. IEEE Transactions on Computers. C-24(6):604-616, June, 1975.

[Davi93] Davis A., Practical Asynchronous Design. Proceedings of the VII Banff Workshop: Asynchronous Hardware Design, Banff, Canada, 1993.

[Davi94] Davies R. Powertool Manual, AMULET group report, Manchester University, 1994

[Day92] Day P., A Micropipelined Multiplier. ACiD-WG/EXACT Workshop on Asynchronous Data Processing. Veldhoven, The Netherlands, December, 1992.

[Day93] Day P., Investigations into Micropipeline Latch Design Styles, (to be published), University of Manchester, 1993.

[DEC92] Alpha Architecture Handbook. Digital Equipment Corporation, Maynard, MA, U.S.A, 1992.

[DEC92a] DECChip 21064-AA RISC Microprocessor Preliminary Data Sheet. Digital Equipment Corporation, Maynard, MA, U.S.A, 1992.

[Dijk76] Dijkstra E.W., A Discipline of Programming. Prentice-Hall, Englewood Cliffs NJ, 1976.

[Dill89] Dill D.L., ACM Distinguished Dissertations: Trace Theory for Automatic Hierachical Verification of Speed-Independent Circuits. MIT Press, 1989.

[Dobb92] Dobberpuhl D. et al, A 200 MHz 64b Dual-Issue CMOS Microprocessor. IEEE Journal of Solid-State Circuits, 27(11):155-1565, November, 1992.

[Eber91] Ebergen J.C., A Formal Approach to Designing Delay-Insensitive Circuits. Distributed Computing. 5(3):107-119, 1991.

[Eber92] Ebergen J.C., Peeters A.M.G., Modulo-N Counters: Design and Analysis of Delay-Insensitive Circuits. 2nd Workshop on Designing Correct Circuits, Lyngby, editors J. Staunstrup, R. Sharp, pp. 27-46. Elsevier Science Publishers, June, 1992.

[Edwa92] Edwards D.A. Pitfalls in Asynchronous Design. AMULET Group Internal Report, 1992.

[Edwa93] Edwards D.A., Design and Implementation of I2C. ESPRIT 6143 - EXACT No: EXACT/MU/93Sept/C.1-Deliverable, Confidential, 1993.

[Ende93] Endecott P.B., Processor Architectures for Power Efficiency and Asynchronous Implementation. MSc Thesis, Dept. of Computer Science, University of Manchester, 1993.

[Eshr89] Eshraghian K., Weste N. Principles of CMOS Design A Systems Perspective. Addison-Wesley, Wokingham, England, 1989.

[Farn93] Farnsworth C., A Brief Report on Handshake Circuits. ACID-WG Report, Manchester, 1993.

[Farn93a] Farnsworth C., AMULET group approach to design examples. ACID-WG workshop on Digital Signal Processing, Barcelona, 1993.

[Farn94] Farnsworth C., Low Power Implementations of an I^2C-Bus Expander, MSc Thesis, Dept. of Computer Science, Manchester University, 1994, to be submitted.

[Furb89] Furber S.B., VLSI RISC Architecture and Organization. Marcel Dekker, New York, 1989.

[Furb92] Furber S.B. Micropipelines - A Case Study. ACiD-WG/EXACT Workshop on Asynchronous Data Processing. Veldhoven, The Netherlands, December, 1992.

[Furb93a] Furber S.B, AMULET1 - An Asynchronous ARM Processor. Symposium Record of Hot Chips V, Stanford University, CA, U.S.A., August, 1993.

[Furb93b] Furber S.B., Day P., Garside J.D., Paver N.C., Woods J.V., A Micropipelined ARM. Proceedings of VLSI ‘93, Grenoble, France, September 1993, best paper award.

[Furb93c] Furber S.B., Computing without Clocks. Proceedings of the VII Banff Workshop: Asynchronous Hardware Design, Banff, Canada, 1993.

[Furb94] Furber S.B., Day P., Garside J.D., Paver N.C., Woods J.V., “AMULET1: A Micropipelined ARM”, IEEE CompCon ‘94, San Francisco, March 1994.

[Gars92] Garside J.D. Micropipeline Structures. ACiD-WG/EXACT Workshop on Asynchronous Data Processing. Veldhoven, The Netherlands, December, 1992.

[Gars93] Garside J.D. A CMOS VLSI Implementaion of an Asynchronous ALU. Proceedings of the IFIP working conference on Asynchronous Design Methodologies, Manchester, England, 1993.

[Gopa90] Gopalakrishnan G., Jain P., Some Recent Asynchronous System Design Methodologies. Technical Report Number UU-CS-TR-90-016, Dept. of Computer Science, University of Utah, October, 1990.

[Gopa92] Gopalakrishnan G., Liebchen A., Dynamic Reordering of High Latency Transactions Using a Modified Micropipeline. 1992 IEEE International Conference on Computer Design: VLSI in Computers & Processors, pp. 336-340. October, 1992.

[Gopa93] Gopalakrishnan G., Some Unusual Micropipeline Circuits. Technical Report UU-CS-TR-93-015, Dept. of C.S., Univ. of Utah, July, 1993.

[Gopa93a]	Gopalakrishnan G., Akella, V., Specification, Simulation, and Synthesis of Self-Timed Circuits. 26th Hawaii Int. Conference on System Science (HICSS 1993), pp. 399-408. January, 1993.
[Hamb92]	Hamburgen W.R., Fitch J.S, Packaging a 150W Bipolar Microprocessor. DEC Western Research Laboratory Research Report 92/1, 1992.
[Hauc93]	Hauck S., Asynchronous Design Methodologies: An Overview. Dept. of Computer Science and Engineering, Technical Report 93-05-07, University of Washington, U.S.A. 1993.
[Hoar78]	Hoare C.A.R, Communicating Sequential Processes. Communications of the ACM, 21(8):666-677, 1978.
[Hoar85]	Hoare C.A.R., Communicating Sequential Processes. Prentice-Hall, 1985.
[Hors89]	Horstmann J.U., Eichel H.W., Coates R.L. Metastability Behavior of CMOS ASIC Flip-Flops in Theory and Test. IEEE Journal of Solid-State Circuits. 24(1):146-157, February, 1989.
[Howa89]	Howard D.W. *Dynamic PLAs*. ARM Ltd internal document, August, 1989.
[Huff64]	Huffman D.A., The Synthesis of Sequential Switching Circuits. Sequential Machines: Selected Papers, editor Moore, E. F., Addison-Wesley, 1964.
[Inmo83]	Occam Programming Manual. Inmos, Bristol, England, 1983.
[Jagg90]	Jagger D.V., A Performance Study of the Acorn RISC Machine. M.Sc. Thesis, University of Canterbury, New Zealand, 1990.
[Jose90]	Josephs M.B., Udding, J.T., Delay-Insensitive Circuits: An Algebraic Approach to their Design. Lecture Notes in Computer Science. editors J.C.M. Baeten and J.W. Klop, Volume 458, pp. 342-366. Springer-Verlag, 1990.
[Jose90a]	Josephs M.B., Mak, R.H., Verhoeff, T., Asynchronous Design of a Router. Dept. of Math. and C.S., Eindhoven Univ. of Technology, September, 1990.
[Jose92]	Josephs M.B. Speed-Independent Design of a Toggle. ACiD-WG/EXACT Workshop on Asynchronous Controllers and Interfacing. Leuven, Belgium, September, 1992.
[Jose91]	Josephs M.B,. Udding, J.T. An Algebra for Delay-Insensitive Circuits. Workshop on Computer-Aided Verification, editors R.Kurshan and E. M. Clarke, pp. 147-175. AMS-ACM, 1991.
[Jose92a]	Josephs M.B., et al, High-level Design of an Asynchronous Packet-routing Chip. In Designing Correct Circuits, editors J. Staunstrup and R. Sharp, IFIP transactions A:Computer Science and Technology, pp. 261-274, North Holland, 1992.

[Joup93] Jouppi N.P. et al, A 300MHz 115W 32b Bipolar ECL Microprocessor. Symposium Record of Hot Chips V, Stanford University, CA, U.S.A., August, 1993.

[Kell74] Keller R.M., Towards a Theory of Universal Speed-Independent Modules. IEEE Transactions on Computers C-32(1):21-33, June 1974.

[Kern88] Kernighan B.W., Ritchie D.M., The C Programming Language. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[Kell93] Kelly R., Asynchronous Design Aspects of High Performance Logic. MSc Thesis in preparation, 1993.

[Kess90] Kessels J.L.W., Schalij F.D., VLSI Programming for the Compact Disc Player. Science of Computer Programming, 15:235-248, 1990.

[Kess90a] Kessels J.L.W., Rem M., Designing systolic, distributed buffers with bounded response time. Distributed Computing, 4:37-43, 1990.

[Kess91] Kessels J.L.W., The Systematic Design of a Systolic RSA Converter. Proceedings of the Workshop on Correct Hardware Design Methodologies, pp. 243-260, 1991.

[Kess92] Kessels J.L.W. et al, An Error Decoder for the Compact Disc Player as an Example of VLSI Programming. Proceedings of the European Conference on Design Automation, pp.69-74, 1992.

[Kinn76] Kinniment D.J., Woods J.V., Synchronisation and arbitration circuits in digital systems. Proc. IEE. 123(10):961-966, October, 1976.

[Lava92] Lavagno L., Sangiovanni-Vincentelli A., Linear Programming for Optimum Hazard Elimination in Asynchronous Circuits. 1992 IEEE International Conference on Computer Design: VLSI in Computers & Processors, pp. 275-278. October, 1992.

[Lin91] Lin K.J., Lin C.S., Automatic Synthesis of Asynchronous Circuits. Proceedings of DAC, pp. 296-301, 1991

[Lin91a] Lin K.J., Lin C.S., A Realization Algorithm of Asynchronous Circuits from STG. Proceedings of EDAC, pp.322-326, 1992

[Lin91a] Lin K.J., Lin C.S.,On the Verification of State-Coding in STGs, Proceedings of ICCAD, pp.118-122, 1992.

[Lind92] Linerholm, O., Aprika, S., Nadeau, M., The PC gets more personal. Byte, pp 128-136, July 1992.

[Mart85] Martin A.J., Distributed Mutual Exclusion on a Ring of Processes. Science of Computer Programming, 5:265-276, 1985.

[Mart85a] Martin A.J., The design of a Self-timed circuit for Distributed Mutual Exclusion. 1985 Chapel Hill Conference on VLSI, editor H. Fuchs, Computer Science Press, pp. 247-260, 1985.

[Mart85b] Martin A.J., A Delay-insensitive Fair Arbiter. Technical Report 5193:TR:85, Computer Science Dept, Caltech, 1985.

[Mart86] Martin A.J., Compiling Communicating Processes into Delay-insensitive VLSI circuits. Distributed Computing, 1(4), 1986.

[Mart89] Martin A.J., Formal Program Transformations for VLSI Circuit Synthesis. UT Year of Programming Institute on Formal Developments of Programs and Proofs, editor E.W. Dijkstra, Addison-Wesley, Reading MA, 1989.

[Mart89a] Martin A.J., Burns S. M., Lee T. K., Borkovic D., Hazewindus P. J., Design of an Asynchronous Microprocessor. Advanced Research in VLSI 1989: Proceedings of the Decennial Caltech Conference on VLSI, ed. C. L. Seitz, MIT Press, pp 351-373, 1989.

[Mart89b] Martin A.J., Burns S. M., Lee T. K., Borkovic D., Hazewindus P. J., The first asynchronous microprocessor. Technical report CS-TR-89-06, Caltech, 1989.

[Mart90] Martin A.J. Synthesis of Asynchronous VLSI Circuits. Formal Methods for VLSI Design, editor J. Staunstrup, North-Holland, 1990.

[Mart90a] Martin A.J., The Limitations to Delay-Insensitivity in Asynchronous Circuits. Sixth MIT Conference on Advanced Research in VLSI, editor W. J. Dally, pp. 263-278. MIT Press, 1990.

[Mead80] Mead C., Conway L. Introduction to VLSI Systems. Addison-Wesley, London, 1980.

[Mehr92] Mehra R., Micropipelined Cache Design Strategies for an Asynchronous Microprocessor. MSc Thesis, Dept. of Computer Science, University of Manchester, 1992.

[Meng89] Meng T.H.Y., Brodersen R.W., Messerschmidtt D.G., Automatic Synthesis of Asynchronous Circuits from High-Level Specifications. IEEE Transactions on CAD, 8(11):1185-1205, November, 1989

[Mill65] Miller R.E. Sequential Circuits. Chapter 10, In Switching Theory, Wiley, New York, 1965.

[Moln83] Molnar C.E., Fang T-P., Synthesis of Reliable Speed-Independent Circuit Modules: I. General Method for Specification of Module-Environment Interaction and Derivation of a Circuit Realization. Technical Report 297, Computer Systems Laboratory, Institute for Biomedical Computing, Washington Univ., St. Louis, MO, 1983.

[Moln85] Molnar C.E., Fang T-P., Rosenberger F.U., Synthesis of Delay-Insensitive Modules. 1985 Chapel Hill Conference on Very Large Scale Integration, editor H. Fuchs, pp. 67-86. Computer Science Press, 1985.

[Myer92] Myers C., Meng T.H-Y., Synthesis of Timed Asynchronous Circuits. 1992 IEEE International Conference on Computer Design: VLSI in Computers & Processors, pp. 279-284. October, 1992.

[Myer93]	Myers C., and Meng T.H-Y., Synthesis of Timed Asynchronous Circuits. 1993. (to appear in) IEEE Trans. on VLSI Systems, June '93.
[Nage73]	Nagel L.W., Pederson D.O., Simulation Program with Integrated Circuit Emphasis (SPICE). Report ERL-M383, University of California, Berkeley, Electronics Research Lab., 1973.
[Nies88]	Niessen C., van Berkel C.H., Rem M., Saeijs R.W.J.J., VLSI Programming and Silicon Compilation: A Novel Approach from Philips Research. Proceedings of ICCD, IEEE, pp. 150-151, 1988.
[Norw91]	Nowick S.M., Dill D.L., Synthesis of Asynchronous State Machines Using a Local Clock. Proc. of 1991 Int. Conf. on Computer Design, pp. 192-197. IEEE Computer Society Press, October, 1991.
[Norw91b]	Nowick S.M., Dill D.L., Automatic Synthesis of Locally-Clocked Asynchronous State Machines. Proc. of 1991 Int. Conf. on Computer-Aided Design, pp. 318-321. IEEE Computer Society Press, November, 1991.
[Norw92]	Nowick S.M., Dill D.L., Asynchronous State Machine Synthesis Using a Local Clock. Proc. of 1991 MCNC Int. Workshop on Logic Synthesis. 1991.
[Pave91]	Paver N.C., Condition Detection in Asynchronous Pipelines. UK Patent no 9114513, October 1991.
[Pave92a]	Paver N.C., Day P., Furber S.B., Garside J.D., and Woods J.V., Register Locking in an Asynchronous Microprocessor. Proceedings of ICCD ‘92, pp 351-355, October 1992.
[Pave92b]	Paver N.C., Micropipelines - Implementation. ACiD-WG/EXACT Workshop on Asynchronous Data Processing. Veldhoven, The Netherlands, December, 1992.
[Pave93]	Paver N.C., Day P., A Micropipelined ARM. Unpublished presentation, IFIP working conference on Asynchronous Design Methodologies, Manchester, England, 1993
[Pete81]	Peterson J., Petri net theory and modeling of systems. Prentice Hall, 1981.
[Rem90]	Rem M., The Nature of Delay-Insensitive Computing. {IV} Higher Order Workshop, Banff 1990, pp. 105-122. Springer-Verlag, 1991.
[Saei88]	Saeijs R.W.J.J., van Berkel C.H., The Design of the VLSI Image-Generator Zap. Proceedings of ICCD, IEEE, pp. 163-166, 1988.
[Seit70]	Seitz C.L., Asynchronous Machines Exhibiting Concurrency. Record of the Project MAC Concurrent Parallel Computation. 1970.
[Seit80]	Seitz C.L., System Timing. In Introduction to VLSI Systems, editors Mead C. A., Conway L. A., chapter 7, Addison-Wesley, 1980.
[Shoj88]	Shoji M. CMOS Digital Circuit Technology. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[Smit92]	Smith L., A Guide to the ASIM Simulation and Modelling Package, ARM Ltd., 1992.
[Stal92]	Stallman R.M., Using and Porting GNU CC. Free Software Foundation Inc., Cambridge, MA, U.S.A, 1992.
[Sun92]	The SuperSPARC Microprocessor. Sun Microsystems Inc., Technical white paper, Moutain View, CA, U.S.A, 1992.
[Suth89]	Sutherland I.E., Micropipelines. Communications of the ACM. 32(6):720-738, January, 1989.
[Suth86]	Sutherland I.E., Sproull R.F. Asynchronous Systems. Sutherland, Sproull & Associates, Palo Alto, California, U.S.A, September, 1986.
[Sven89]	Svensson C., Yuan J., High-Speed CMOS circuit techniques. IEEE Journal of Solid-State Circuits, 24(1):62-72, February, 1989.
[Theo93]	Theodoropoulos G.K., Woods J.V., Modeling and Simulation of Asynchronous Computer Architectures. (to be published), Manchester University, 1993.
[TTL85]	The TTL Data Book: Volume 1. Texas Instruments, 1985.
[Unge59]	Unger S.H., Hazards and delays in asynchronous sequential switching circuits, IRE Trans. Circuit Theory, CT-6:12-25, March, 1959.
[Unge69]	Unger S.H., Asynchronous Sequential Switching Circuits. Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.
[Vanb90b]	Vanbekbergen P. et al, Optimized Synthesis of Asynchronous Control Circuits from Graph-theoretic Specifications. Proceedings of ICCAD, pp.184-187, 1990.
[Vanb90b]	Vanbekbergen P. et al, Time & Area Perfomant Synthesis of Asynchronous Control Circuits. Proceedings of TAU'90, 1990.
[vBer88]	van Berkel C.H., Rem M., Saeijs R.W.J.J., VLSI Programming. Proceedings of ICCD, IEEE, pp. 152-156, 1988.
[vBer88a]	van Berkel C.H., Rem M., Saeijs R.W.J.J., Compilation of Communicating Processes into Delay-Insensitive Circuits. Proceedings of ICCD, IEEE, pp. 157-162, 1988.
[vBer91]	van Berkel C.H., Kessels J., Roncken M., Saeijs R.W.J.J, Schalij F., The VLSI-programming langauge Tangram and its translation into handshake circuits. Proceedings of EDAC, pp. 384-389, 1991.
[vBer92]	van Berkel C.H., Handshake circuits: an intermediary between communicating processes and VLSI, PhD thesis, Eindhoven Univeristy of Technology, 1992.
[vBer92a]	van Berkel C.H., Beware the Isochronic Fork. INTEGRATION, the VLSI Journal. 13(2):103-128, June, 1992.
[VLSI90]	Acorn Risc Machine (ARM) Family Data Manual. VLSI Technology Inc., Prentice Hall, Englewood Cliffs, New Jersey, 1990

[VLSI91] CMOS Lamda Layout Design Rules for Advanced Lithography. VLSI Technology Inc., San Jose, U.S.A., Document No: 02-ECLA-2, Confidential, December, 1991.

[Will90] Williams T.E., Latency and Throughput Tradeoffs in Self-Timed Asynchronous Pipelines and Rings. Technical Report CSL-TR-90-431, Computer Systems Laboratory, Stanford University, August, 1990.

[Will91] Williams T.E., Horowitz M.A., A 160ns 54bit {CMOS} Division Implementation using Self-Timing and Symmetrically Overlapped {SRT} Stages. Proceedings of the 10th {IEEE} Symposium on Computer Arithmetic, pages 210-217. 1991

[Will91a] Williams T.E., Horowitz M.A., A Zero-Overhead Self-Timed 160-ns 54-b CMOS Divider. IEEE Journal of Solid-State Circuits. 26(11):1651-1661, November, 1991.

[XIDA87] Mainsail Language Manual. Volume 1, XIDAK Inc., Menlo Park, CA, U.S.A. 1987.

[Yako92] Yakovlev A.V., On Limitations and Extensions of STG Model for Designing Asynchronous Control Circuits. Proceedings of ICCD, pp. 396-400, 1992.

[Yant92] Yantchev J., An S-I Toggle Design. Private fax communication, 1992.

[Yun92] Yun K.Y., Dill D.L., Nowick S.M., Synthesis of 3D Asynchronous State Machines. 1992 IEEE International Conference on Computer Design: VLSI in Computers & Processors, pp. 346-350. October, 1992.

[Yun92b] Yun K.Y., Dill D.L., Nowick, S.M., Practical Generalizations of Asynchronous State Machines. Technical Report CSL-TR-92-544, Computer Systems Laboratory, Stanford University, July, 1992.

Appendix A: Timing characteristics

This appendix contains the timing characteristics of some of the Micropipeline library elements designed for use in the asynchronous ARM. Each cell is characterised by simulating the extracted netlist with SPICE under worst case conditions for a selection of loads. The exact conditions used are described in the next section. The cells analysed are listed below:

CALL2	- 2 input call block
DMULLC2	- 2 input C-Gate with double strength drive on the input stack to enable the gate to drive from the internal node
DXOR/DXNOR	- 2 input XOR, each input requires complimentary
DWAIT2	- 2 input Decision-Wait
MULLC2R	- 2 input C-Gate with reset
MULLC2	- 2 input C-Gate without reset
MULLC3R	- 2 input C-Gate with reset
TOGGLE	- TOGGLE element
SELECT2	- 2 input SELECT block
TLTCHR	- transparent latch with true & complement enable required

A.1 Measurement conditions

The measurements are taken at the following conditions:

VDD	**4.6**	**VSS**	**0.1**
Temp	**100^{o}C**		
Process	**CMOS 1.2 micron, Slow-Slow transistors**		

All measurements in the tables are in nanoseconds (unless otherwise stated)

Inputs ramp 0-100% (0-4.6v) in 4 nS

The propagation delay is measured from the 50% point of the input to the 50% point on the output.

The rise/fall time is measured between the 10% and 90% value of the output.

SPICE timings for: CALL2

Change history

version	**author**	**date**	**comment**
0.1	N.C.Paver	5/5/1993	initial version

Signals

There are three input signals *R1*, *R2*, *D* and the complements *nR1* and *nR2*. There is also a reset *Cdn*. There are three outputs *D1*, *D2*, *R* and their complements *nD1* and *nD2.* The forward measurement (*R1/R2 -> R)* is equivalent to a DXOR and the reverse measurement (*D -> D1/D2)* is the same as a DWAIT2. The measurements shown below are taken from the appropriate corresponding gate.

D -> D1/ D2 (equal load on *D1/nD1* and *D2/nD2)*

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	3.51	4.22	1.81	1.39
0.12	2	3.97	4.73	2.32	1.76
0.18	4	4.40	5.21	2.85	2.15
0.24	5	4.83	5.68	3.37	2.53
0.50	10	6.70	7.69	5.71	4.23
1.00	20	10.08	11.54	10.17	7.42

D- > nD1/nD2 (equal load on *D1/nD1* and *D2/nD2*)

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	3.11	2.64	2.59	2.18
0.12	2	3.28	2.89	3.03	2.51
0.18	4	3.45	3.07	3.49	2.81
0.24	5	3.60	3.28	3.91	3.11
0.50	10	4.29	4.19	5.92	4.72
1.00	20	5.54	5.71	9.78	7.56

R1/R2 -> R

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	0.63	1.31	2.01	1.48
0.12	2	0.98	1.68	2.74	1.96
0.18	4	1.31	2.05	3.52	2.47
0.24	5	1.57	2.42	4.29	2.86

R1/R2 -> R

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.50	10	2.50	4.10	7.95	4.65
1.00	20	4.25	7.28	15.18	8.51

Capacitance

R	**0.071PF**
R1	**0.145PF**
R2	**0.186PF**
nR1/nR2	**0.05PF**
D	**0.170PF**
D1/D2	**0.184PF**
nD1/nD2	**0.275PF**
Cdn	**0.227PF**

SPICE timings for: DMULLC2

Change history

version	**author**	**date**	**comment**
0.1	N.C.Paver	5/5/1993	initial version

Signals

There are 2 input signals *In1*, *In2*, and the reset signal *Cdn*. There are two outputs *Out* and *nOut*. The measurements are for equal loads on both outputs.

Out

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	2.39	2.09	1.71	1.28
0.12	2	2.86	2.57	2.11	1.54
0.18	4	3.28	3.06	2.61	1.94
0.24	5	3.73	3.55	3.13	2.31
0.50	10	5.55	5.61	5.46	3.88
1.00	20	9.10	9.50	9.73	6.98

nOut

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	1.14	1.73	2.78	2.01
0.12	2	1.32	1.96	3.68	2.32
0.18	4	1.49	2.14	3.60	2.69
0.24	5	1.66	2.35	3.95	2.99
0.50	10	2.32	3.18	5.85	4.35
1.00	20	3.61	4.83	9.44	7.23

Capacitance

In1/In2	**0.09PF**
Out	**0.134PF**
nOut	**0.226PF**
Cdn	**0.105PF**

SPICE timings for: DXor/DXNor

Change history

version	**author**	**date**	**comment**
0.1	N.C.Paver	5/5/1993	initial version

Signals

There are 4 input signals *In1, In2, nIn1, nIn2*. There is a single output *Out*.

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	0.63	1.31	2.01	1.48
0.12	2	0.98	1.68	2.74	1.96
0.18	4	1.31	2.05	3.52	2.47
0.24	5	1.57	2.42	4.29	2.86
0.50	10	2.50	4.10	7.95	4.65
1.00	20	4.25	7.28	15.18	8.51

Capacitance

In1/In2	**0.038PF**

SPICE timings for: DWAIT2

Change history

version	author	date	comment
0.1	N.C.Paver	5/5/1993	initial version

Signals

There are three input signals *A1*, *A2*, *Fire* and the reset *Cdn*. There are two outputs *Z1* and *Z2* and their complements *nZ1* and *nZ2.* The measurements are taken with equal loads on true and complement outputs.

Z1/Z2

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	3.51	4.22	1.81	1.39
0.12	2	3.97	4.73	2.32	1.76
0.18	4	4.40	5.21	2.85	2.15
0.24	5	4.83	5.68	3.37	2.53
0.50	10	6.70	7.69	5.71	4.23
1.00	20	10.08	11.54	10.17	7.42

nZ1/nZ2

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	3.11	2.64	2.59	2.18
0.12	2	3.28	2.89	3.03	2.51
0.18	4	3.45	3.07	3.49	2.81
0.24	5	3.60	3.28	3.91	3.11
0.50	10	4.29	4.19	5.92	4.72
1.00	20	5.54	5.71	9.78	7.56

Capacitance

A1/A2	**0.098PF**
Z1/Z2	**0.183PF**
nZ1/nZ2	**0.274PF**
Cdn	**0.226PF**

SPICE timings for: MULLC2R

Change history

version	**author**	**date**	**comment**
0.1	N.C.Paver	5/5/1993	initial version

Signals

There are 2 input signals *In1, In2,* and the reset signal *Cdn*. There are two outputs *Out* and *nOut.* Only the true output should be used as the complimentary output has poor drive capabilities.

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	2.31	2.02	1.61	1.42
0.12	2	2.51	2.24	2.06	1.68
0.18	4	2.72	2.46	2.56	1.78
0.24	5	2.87	2.69	2.92	1.92
0.50	10	3.53	3.61	4.79	2.99
1.00	20	4.72	5.37	8.59	5.20

Capacitance

In1/In2	**0.05PF**
Out	**0.129PF**
Cdn	**0.054PF**

SPICE timings for: MULLC2

Change history

version	**author**	**date**	**comment**
0.1	N.C.Paver	5/5/1993	initial version

Signals

There are 2 input signals *In1*, and *In2*. There are two outputs *Out* and *nOut.* Only the true output should be used as the complimentary output has poor drive capabilities.

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	2.13	1.62	1.48	1.36
0.12	2	2.33	1.86	1.92	1.60
0.18	4	2.53	2.09	2.26	1.78
0.24	5	2.68	2.30	2.77	2.01
0.50	10	3.35	3.24	4.75	2.97
1.00	20	4.52	4.99	8.57	5.12

Capacitance

In1/In2	**0.05PF**
Out	**0.129PF**
Cdn	**0.054PF**

SPICE timings for: MULLC3R

Change history

version	**author**	**date**	**comment**
0.1	N.C.Paver	5/5/1993	initial version

Signals

There are 3 input signals *In1*, *In2*, *In3,* and the reset signal *Cdn*. There are two outputs *Out* and *nOut.* Only the true output should be used as the complimentary output has poor drive capabilities.

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	2.88	2.27	1.64	1.34
0.12	2	3.11	2.53	2.11	1.64
0.18	4	3.33	2.77	2.59	1.96
0.24	5	3.53	3.01	3.02	2.29
0.50	10	4.28	3.98	5.03	3.42
1.00	20	5.54	5.76	8.71	5.60

Capacitance

In1/In2/In3	**0.05PF**
Out	**0.129PF**
Cdn	**0.054PF**

SPICE timings for: TOGGLE

Change history

version	author	date	comment
0.1	N.C.Paver	5/5/1993	initial version

Signals

There is a single input *In* and the reset signal *Cdn*. There are two outputs *Dot* and *Blank and* their complements *nDot* and *nBlank.* The measurements are for equal loads on both true and complement outputs.

Dot

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	2.97	2.52	1.62	1.26
0.12	2	3.42	3.07	2.00	1.59
0.18	4	3.98	3.61	2.47	1.97
0.24	5	4.39	4.16	3.00	2.28
0.50	10	6.63	6.49	5.15	3.80
1.00	20	10.74	10.89	9.06	7.01

nDot

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	1.46	2.19	2.87	2.80
0.12	2	1.70	2.44	3.42	3.21
0.18	4	1.95	2.75	4.15	3.64
0.24	5	2.19	3.00	4.62	4.06
0.50	10	3.21	4.31	7.39	6.27
1.00	20	5.19	6.64	12.61	10.6

Blank

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	2.57	3.58	1.43	1.06
0.12	2	3.06	4.13	1.80	1.50
0.18	4	3.57	4.66	2.40	1.86
0.24	5	4.10	5.16	2.70	2.27
0.50	10	6.17	7.47	4.66	3.83

Blank

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
1.00	20	10.32	11.97	8.62	7.04

nBlank

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	2.62	1.96	3.44	2.78
0.12	2	2.93	2.22	3.97	3.25
0.18	4	3.12	2.58	4.62	3.77
0.24	5	3.36	2.86	5.33	4.19
0.50	10	4.41	4.02	7.90	6.38
1.00	20	6.45	6.46	13.12	10.95

Measurements taken with only *Dot* and *Blank* loaded (Temp=0C):

Dot

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	2.61	2.19	1.60	1.10
0.12	2	2.71	2.35	1.93	1.32
0.18	4	2.89	2.52	2.22	1.54
0.24	5	3.06	2.70	2.54	1.66
0.50	10	3.58	3.40	3.94	2.64
1.00	20	4.50	4.73	6.69	4.42

nDot

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	1.17	1.89	2.35	2.30
0.12	2	1.17	1.84	2.27	2.20
0.18	4	1.17	1.86	2.02	2.17
0.24	5	1.17	1.91	2.00	2.17
0.50	10	1.17	1.91	1.95	2.17
1.00	20	1.17	1.84	1.92	2.17

Blank

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	2.24	3.22	1.19	1.19

Blank

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.12	2	2.43	3.44	1.54	1.22
0.18	4	2.58	3.69	1.92	1.43
0.24	5	2.72	3.80	2.19	1.63
0.50	10	3.28	4.51	3.53	2.54
1.00	20	4.28	5.92	6.32	4.42

nBlank

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	2.41	1.69	2.88	2.25
0.12	2	2.41	1.69	2.90	2.28
0.18	4	2.43	1.66	2.88	2.28
0.24	5	2.41	1.64	2.85	2.15
0.50	10	2.39	1.69	2.86	2.06
1.00	20	2.46	1.69	2.86	2.23

Capacitance

In	**0.213PF**
Dot	**0.182PF**
nDot	**0.176PF**
Blank	**0.109PF**
nBlank	**0.232PF**
Cdn	**0.186PF**

SPICE timings for: SELECT2

Change history

version	author	date	comment
0.1	N.C.Paver	5/5/1993	initial version

Signals

There are 2input signals *In1* and *Select* and the reset signal *Cdn*. There are two outputs *True* and *False and* their complements *nTrue* and *nFalse.* The measurements are for equal loads on both true and complement outputs.

True/False

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	3.85	4.73	1.86	1.46
0.12	2	4.63	5.71	2.44	1.85
0.18	4	5.40	6.70	3.09	2.22
0.24	5	6.16	7.63	3.47	2.62
0.50	10	9.52	11.83	6.06	4.24
1.00	20	15.64	19.71	10.30	7.48

nTrue/nFalse

Load	Stand. Load	Fall Prop.	Rise Prop	Rise	Fall
0.06	1	3.56	2.95	2.77	2.36
0.12	2	4.18	3.50	3.26	2.86
0.18	4	4.83	4.05	3.82	3.43
0.24	5	5.48	4.58	4.39	3.97
0.50	10	8.19	6.93	6.75	6.28
1.00	20	13.38	11.17	11.31	10.30

Capacitance

In	**0.168PF**
Sel	**0.181PF**
True/False	**0.194PF**
nTrue/NFalse	**0.278PF**
Cdn	**0.227PF**

SPICE timings for: TLTCHR

Change history

version	**author**	**date**	**comment**
0.1	P.Day	5/5/1993	initial version

Signals

There are 3 input signals *In*, *En, NEn*, and the reset signal *Cdn*. There are two outputs *Out* and *nOut.* It should be noted that the complimentary output has poor drive capabilities.

In -> Out (nOut)

Load	Stand. Load	I/P Fall Prop.	I/P Rise Prop.	Rise	Fall
0.06	1	2.51 (1.84)	2.08 (1.08)	1.61 (2.27)	1.25 (2.98)
0.12	2	2.72 (1.85)	2.32 (1.09)	2.05 (2.15)	1.45 (3.02)
0.18	4	2.90 (1.84)	2.54 (1.09)	2.53 (2.12)	1.69 (3.06)
0.24	5	3.07 (1.84)	2.76 (1.09)	2.97 (2.10)	1.97 (3.06)
0.50	10	3.73	3.69	4.89	3.07
1.00	20	4.79	5.45	8.75	5.17

En/nEn -> Out (nOut)

Load	Stand. Load	En -> O/P low	En -> O/P high	Rise	Fall
0.06	1	2.21 (1.58)	1.79 (0.82)	1.67 (2.11)	1.44 (2.87)
0.12	2	2.41 (1.57)	2.01 (0.82)	2.16 (1.96)	1.72 (2.92)
0.18	4	2.62 (1.58)	2.25 (0.83)	2.54 (1.94)	1.88 (2.95)
0.24	5	2.80	2.49	2.95	2.10
0.50	10	3.48	3.42	4.94	3.06
1.00	20	4.65	5.19	8.78	5.20

Capacitance

In	0.029pF		
En	0.035pF	nEn	0.058pF
Out	0.132pF	nOut	0.143pF
Cdn	0.054pF		

Appendix B: The ARM processor

The Advanced RISC Machine (**ARM**) is a 32-bit general purpose reduced instruction set microprocessor. The 1987 ARM2 will be described first, followed by the modifications that have lead to the ARM6.

B.1 The ARM2

The ARM2 features 27 registers each of 32 bits. The registers are organized into a set of partially overlapping banks of registers. There are fifteen general purpose registers and a register containing both the program counter and the program status register (PSR) available at any time. The ARM2 supports four modes - User, Supervisor (*SVC*), Interrupt (*IRQ*) and Fast Interrupt (*FIQ*). Figure B-1 shows how the registers banks are mapped on to the four modes.

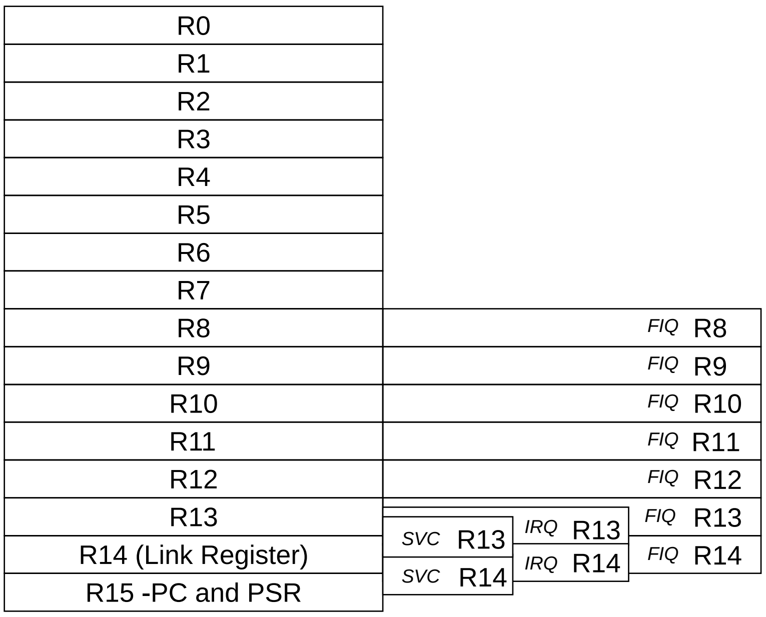


Figure B-1: The ARM2 register organization

The external address bus of the ARM2 is only 26-bits wide. This gives an addressing range of 64 MBytes. A consequence of this is that the program counter needs only to cover this range. Instructions in the ARM2 are always word aligned so in fact only 24-bits are needed for the program counter. The remaining bits in the 32-bit word are used to store the processor status bits. The format of the combined PC and PSR is shown in figure B-2. The

PC and PSR can be treated as a single entity for saving and loading but the PC can be used separately when it is used as a base address.

B.1.1 Instruction set

The ARM2 instruction set is based on a load/store model. There are six classes of instructions. The first is data operations; these perform arithmetic and logical operations on the register contents. The status flags of the processor can be changed according to the result of the operation if the 'set flags' bit in the instruction is true. This allows the programmer to determine which instructions can change the status flags. The ARM2 also allows all the data operations (except multiplies) to shift, by an arbitrary amount, one of the operands before performing the operation so there is no separate shift operation.

The second class of instructions loads and stores data which transfer data between memory and the register bank. The address in memory is calculated from the contents of a register (the base) and the addition of either a second register or a 12-bit immediate value (the offset). The contents of the register containing the offset may optionally be shifted before the address is calculated. The instruction class also supports auto increment/decrement of the base register with the option of specifying whether the change should be before or after the base is used to calculate the current memory address (i.e. pre-/post-increment/ decrement).

The third class of instruction allows multiple registers to be transferred to or from memory by a single instruction. The registers to be transferred are specified by the programmer. The registers are stored to memory in a sequential manner at an address determined by the contents of a base register. The base register is modified after the operation. How the base is modified and where exactly the registers are stored in relation to the base address can be

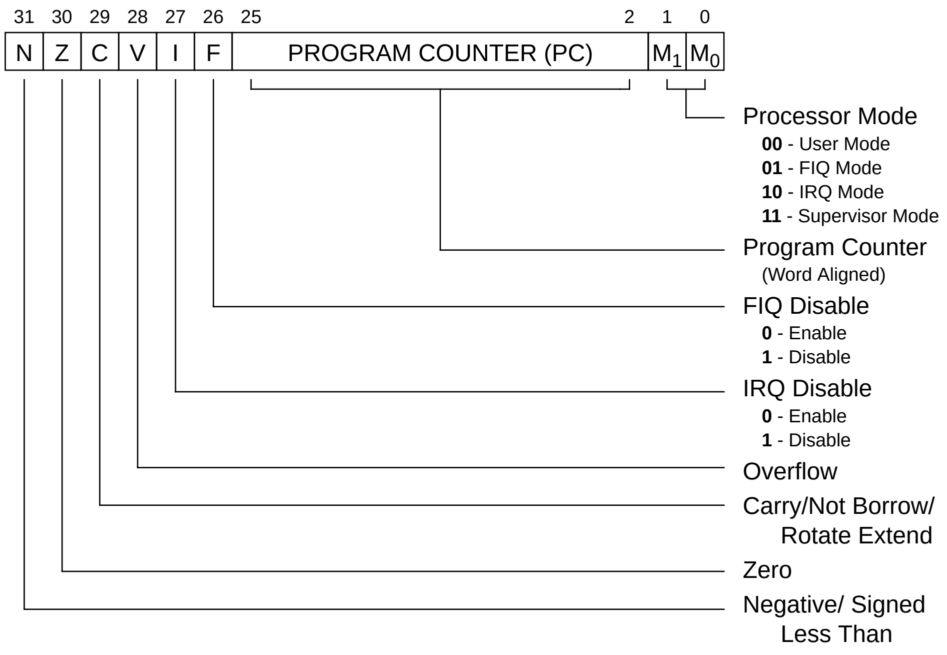


Figure B-2: Program counter and program status word

configured such that various stack paradigms can be supported (e.g. empty/full ascending/descending stacks). These instructions are intended to make procedure entry/exit more efficient (i.e. saving and restoring registers).

The next class of instruction are branches. A special link flag is available to the programmer to cause the address of the following instruction to be placed in the link register (*R14*). This allows subroutine calls to be made. To return from a subroutine entered in this manner, the contents of *R14* are used as the new program counter (i.e. move *R14* -> *R15*).

The remaining instruction classes cover supervisor calls and external coprocessors.

An unusual feature of the ARM2 is that all instructions are conditionally executed, so that short forward branches are usually unnecessary. The instruction set is described in more detail elsewhere [Cock87].

B.1.2 Organization

The internal architecture is 32-bit and is organized as shown in figure B-3. This shows the register bank described previously, the barrel shifter, a 2-bit Booth's multiplier and the ALU. The address incrementer is used to generate the sequential addresses during instruction prefetch and multiple register transfer instructions.The external data bus is also 32-bits but the address bus is only 26-bits thus giving an address range of 64 Mbyte.

The operation of the ARM2 is pipelined into three stages as follows:-

1. Instruction PreFetch
2. Instruction Decode
3. Execute

The execute stage is the complete datapath operation i.e. register read, shift, ALU operation and result writeback. Each stage takes one cycle to complete so that a new instruction may start every cycle.

The ARM2 is described in detail elsewhere [Furb89, VLSI90].

B.2 The ARM6

The ARM6 was developed to extend the 26-bit address range to 32 bits and to provide additional modes to ease operating system design. The two new modes each have a corresponding set of *R13/R14* registers so that the total number of registers has increased from 27 to 31, but still with only sixteen visible at any one time. The register structure is illustrated in figure B-4.

A consequence of the 32-bit program counter is that the processor status information can no longer be stored in the same register. The concept of a separate PSR has been introduced to solve this problem. There is the Current Processor Status Register (CPSR) which contains the working flags and there is a set of Saved Processor Status Registers (SPSRs) one for each of the privileged modes. This allows the status information to be saved across mode changes. Two new instructions have been added to allow the status information to be transferred to and from the CPSR and SPSR (of the current mode) to one of the general purpose registers. The format of the new CPSR/SPSR is similar to the format of the ARM2 status flags, with the addition of 3 extra mode bits and with the I/F flags moved from bits 27/26 (as shown in figure B-2) to bits 7/6.

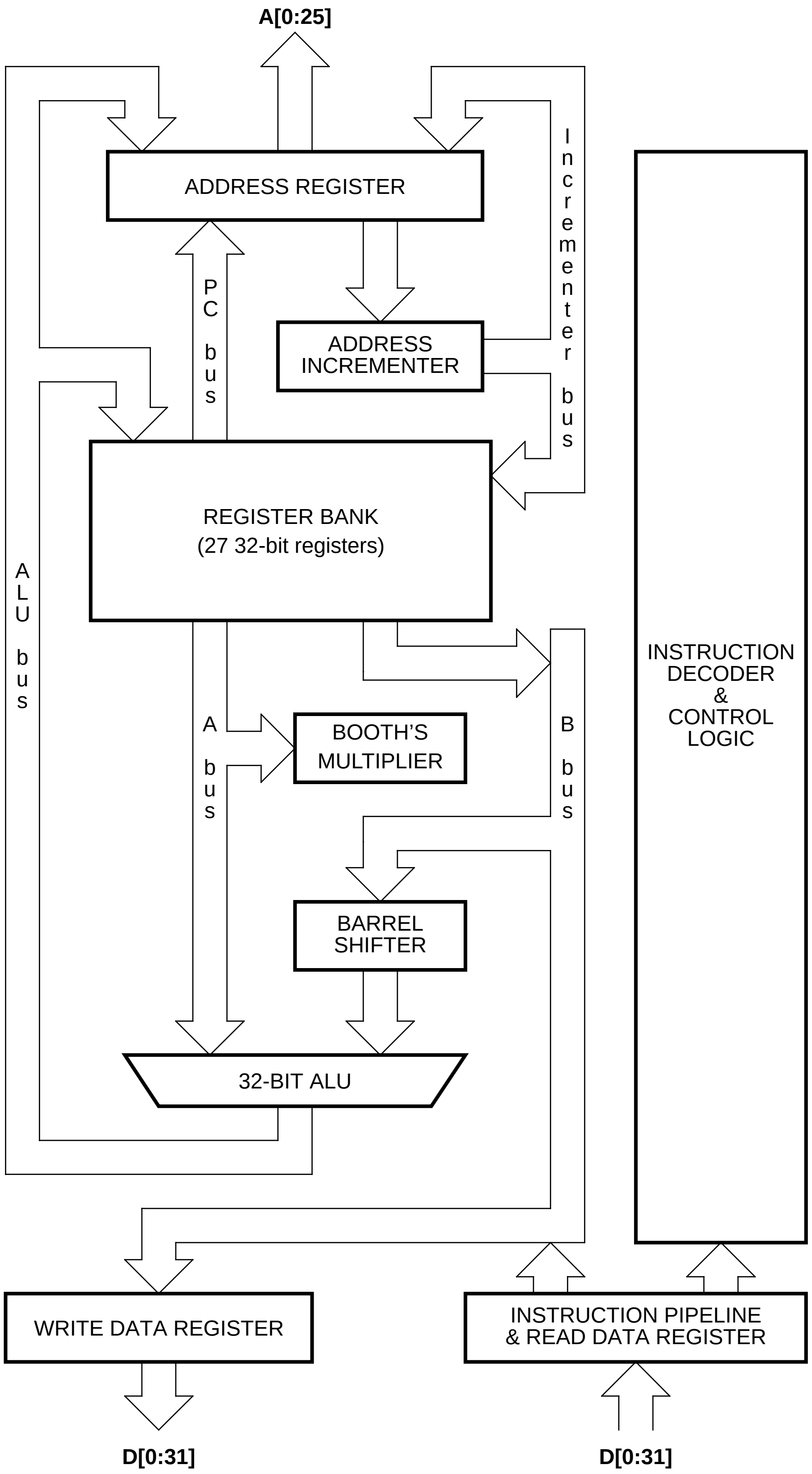


Figure B-3: ARM2 block diagram

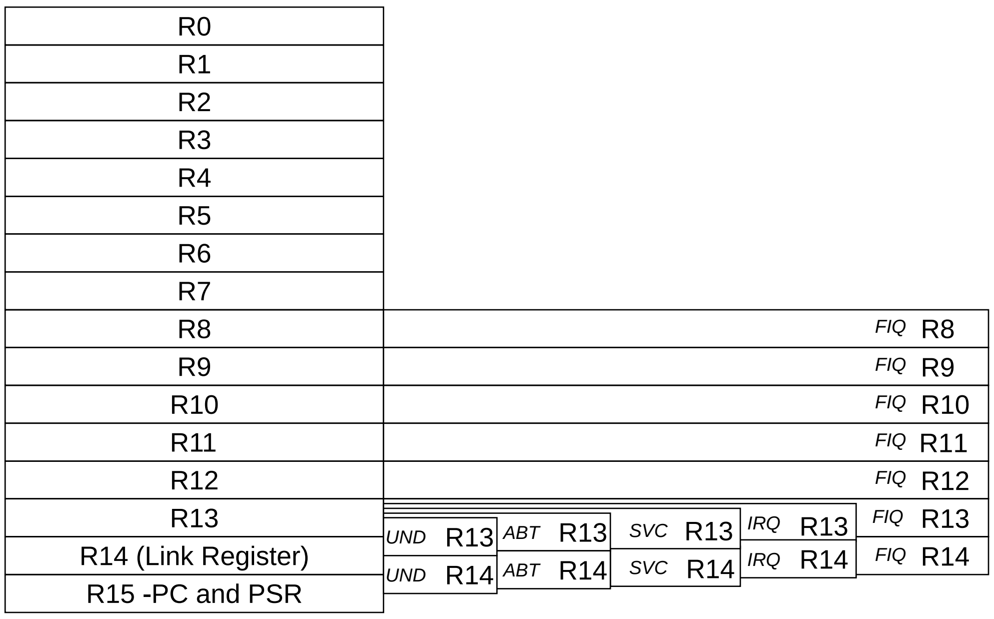


Figure B-4: The ARM6 register organization

The general functionality of the processor remains the same as the ARM2. There is a special mode flag that allows the ARM6 to operate as a 26-bit ARM2. This flag can be set either by strapping a hardware pin or by changing the flag which appears in the CPSR by software. When running in 26-bit mode, the ARM6 has only four modes and 27 registers and a combined PC and PSR (as described in the ARM2 section). The ARM6 is described in more detail elsewhere [ARM91].