Asynchronous Design Aspects of High-Performance Logic

Architectural Modelling of a Bipolar Asynchronous Microprocessor

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

Robert Kelly
Department of Computer Science
1995

Table of Contents

Table of Contents 2
List of Figures 5

1. Introduction 11

2. Asynchronous Logic 14

2.1 Delay Modelling 17
2.2 Signalling Protocols 18
2.2.1 Two-Phase Signalling 19
2.2.2 Four-Phase Signalling 20
2.2.3 Data Communications 22
2.2.4 Bundled-Data Interface 23

3. Micropipelines 25

3.1 Control Circuit Elements 28
3.1.1 XOR (Merge) 28
3.1.2 Muller-C (Join) 28
3.1.3 Select 29
3.1.4 Toggle 29
3.1.5 Decision-Wait 30
3.1.6 Arbiter 30
3.1.7 Call 31
3.1.8 Capture-Pass Latch 32
3.2 Control Circuit Examples 32
3.2.1 Event Register 32
3.2.2 Design Example: PARITY FUNCTION 34

4. Verilog HDL 36

4.1 Introduction to HDLs 36
4.2 Introduction to Verilog 37
4.3 Modules 37
4.4 Structural Modelling 38
4.4.1 Design Example: RS Flip-Flop 39
4.5 Behavioural Modelling 40
4.5.1 Compound Statements 41
4.5.2 Process Control 42
4.5.3 Timing Control 42
4.5.4 Design Example: Behavioural Representation 44
4.5.5 Programmable Logic Arrays 45
4.6 Verilog Simulator 46

5. Multi-Level Differential Current Mode Logic 48
5.1 Introduction to Logic Families 48
5.2 Multi-Level Differential Current Mode Logic 50
6. Verilog Modelling of MDCML 55
6.1 Requirement for Accurate Model of System 55
6.2 Determination of Electrical Characteristics of MDCML 55
6.2.1 Output Loading Effects 58
6.2.2 Input Drive Effects 59
6.3 Production of Verilog Model 61
6.3.1 Accuracy Comparison 63
6.3.2 Continuous Assignment 66
6.3.3 Net Delays 67
7. MDCML Asynchronous ARM 69
7.1 ARM Architecture 69
7.1.1 Overview 69
7.1.2 Instruction Set 71
7.2 MDCML Asynchronous ARM 72
7.2.1 Overview 72
7.2.2 Register Bank 76
7.2.3 Memory Interface 83
7.2.4 Address Interface 83
7.2.5 Data Interface 93
7.2.6 Execution Unit 100
7.2.7 Comments on the MDCML Asynchronous ARM Design 106
8. Architectural Modelling 107
8.1 Introduction 107
8.2 Modelling 107
8.3 Features 113
8.3.1 Instantiation Parameters 113
8.3.2 Test Vector Generation 114
8.4 Code Execution 117
8.4.1 Compilation Method 117
8.4.2 Validation Suite 117
8.4.3 Dhrystone Benchmark 118
8.5 Usage 120
8.5.1 Instrumentation 120
8.5.2 Graphical Output 122
8.5.3 Detecting Incorrect Operation 125
8.6 Performance analysis 126
8.6.1 Subsystem Processing Performance 126

8.6.2 Non-symmetrical Propagation Delays 128
8.6.3 Processor-Memory Interaction 129
8.6.4 Internal Pipeline Efficiency 130
8.6.5 Comments on the Performance Analysis. 133

9. Conclusions 134

9.1 Production of the System Model 134
9.2 Current State of the Project 135
9.3 Comments on the Verilog Modelling Environment 135
9.4 Future Research 137
9.4.1 Technology Migration 137
9.4.2 Architectural Design Alternatives 137

References 139

Appendix A: Verilog Model 142

List of Figures

Two-phase (transition) Signalling. 19
Four-phase Signalling (incorrect operation). 20
Four-phase Signalling (correct operation). 21
Bundled Data Interface. 23
Data Value Constraints. 24
Event Register. 33
Micropipeline Control Block Implementation of Parity Function. 35
Possible Implementation of RS Flip-Flop. 39
Dynamic Power Dissipation. 49
MDCML 3-Input AND Gate. 51
4:1 Multiplexer. 52
Transparent Latch with Reset. 53
2-input AND gate. 56
SPICE model of AND2. 56
2-input AND gate SPICE waveforms. 57
Graphs of Additional Load vs Additional Delay. 59
Graphs of Additional Delay vs Extent of Drive Sharing. 60
Ain after Bin SPICE waveforms. 65
ARM6 Block Diagram. 70
Internal Processor Organisation. 73
Micropipelined Structure of the Execution Pipeline. 75
Register Bank Operation. 76
Asynchronous Register Bank Design. 78
Register Bank Read Cycle Waveform. 80
Register Bank Stalled Read Waveform. 81
Register Bank Write Cycle Waveform. 82
Address Interface Structure. 84
Address Interface Instruction Prefetching Waveform. 89
Address Interface Data Transfer Waveform. 90
Address Interface Block Data Transfer Waveform. 91
Address Interface Branch Waveform. 92
Data Interface Structure. 93
Data Interface Byte Read Waveform. 97
Data Interface Byte Write Waveform. 98
Data Interface Instruction Read Waveform. 99
Execution Unit Structure. 100

Execution Unit Multiply Waveform. 104
Execution Unit Shifted ALU Operand Waveform. 105
MDCML Asynchronous ARM Processor Diagram. 111
MDCML Asynchronous ARM ‘Top-Level’ Verilog Model. 112
Register Bank Display using Verilog gr_regs() system task. 123
Pipeline Occupancy using Verilog gr_bars() system task. 124
Graph of Block Processing Time vs Dhrystone performance. 127
Effect of Non-Symmetrical Propagation Delays. 129
Effect of Memory Speed on Processor Performance. 130
Pipeline Occupancy during Benchmark Execution. 131
Effect of Pipeline Length on Processor Performance. 132

Abstract

As VLSI process technologies develop and feature sizes shrink, the global clocking schemes currently employed in synchronous systems are beginning to experience difficulties in a number of areas. Asynchronous circuits have a potentially higher performance than synchronous circuits since an asynchronous circuit exhibits average-case performance, in contrast to synchronous systems, which must be specifically designed to accommodate worst-case conditions. However, asynchronous design techniques are not widely understood or developed, particularly in the context of a large, complex system.

Recently, an asynchronous design methodology, namely Micropipelines, has been presented which has proved useful in developing an asynchronous CMOS implementation of an existing commercial RISC architecture. A subsequent project has been initiated to develop architectural modelling and implementation tools for an asynchronous high-performance bipolar implementation of the same target architecture.

This thesis presents the issues involved in asynchronous logic design, the details of the particular asynchronous design methodology employed and an introduction to the architectural modelling environment used in the development of the bipolar asynchronous implementation. The development of the system model is illustrated, with reference to the underlying primitive components and the hierarchical composition of the complete design from asynchronous sub-functions communicating via a well-defined signalling protocol. A demonstration of how the architectural model can be used to generate information regarding the internal operation of the system, which is then used to improve the complete design is given. The suitability of modelling asynchronous systems with the modelling environment employed is discussed.

Declaration

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

Copyright in text of this thesis rests with the Author. Copies (by any process) either in full, or of extracts, may be made **only** in accordance with instructions given by the Author and lodged in the John Rylands University Library of Manchester. Details may be obtained from the Librarian. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the Author.

The ownership of any intellectual property rights which may be described in this thesis is vested in the University of Manchester, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the University, which will prescribe the terms and conditions of any such agreement.

Further information on the conditions under which disclosures and exploitation may take place is available from the Head of Department of Computer Science.

Preface

The author was employed in the motor and light engineering industries before returning to full-time study in 1987. He graduated with a B.Sc. (Hons.) Degree in Computer Engineering from the University of Manchester in 1990. The author was previously employed as a Research Associate on the ESPRIT-funded EDS project, being involved in operating system kernel developments for a message-passing multiprocessor architecture. He is currently employed as a Research Associate in the AMULET group at the University of Manchester on the Transforming Architectural Models (TAM-ARM) project.

Acknowledgements

The work presented here was carried out as part of the TAM-ARM project funded within the Design Automation section of the DTI/SERC Advanced Technology Programme. The author gratefully acknowledges this support and the technical assistance provided by the industrial project partners: Advanced RISC Machines (ARM) Limited and GEC Plessey Semiconductors.

I would like to express my sincere thanks to my project supervisor Dr. L.E.M. Brackenbury, for her support and encouragement. I would also like to thank members of the original AMULET1 design team: Prof. Steve Furber, Nigel Paver, Paul Day and Jim Garside for revealing the innermost secrets of their creation.

The AMULET research group in the Computer Science department of the University of Manchester provided a stimulating and interesting environment in which to contemplate Life, the Universe and Asynchronous Processor Design - to the remaining group members, both staff and students, I am indebted to you for making it so.

Finally, to my wife, Kathryn, and daughters, Leah and Laura, who have endured so much over the past few months without complaint (well, almost!), I am eternally grateful.

1. Introduction

"By combining advances in integrated circuit technology, improvements in compiler design and new architectural ideas, significant performance improvements have been realised in the contemporary design of computer systems. These improvements have only been made possible by bringing together important technological advances with a better empirical understanding of how computers are used. From this fusion has emerged a style of computer design based on **empirical data**, **experimentation** and **simulation**."

These ideas, drawn from probably the most important text on computer design over the past decade - Computer Architecture: A Quantitative Approach [Henn90] - indicate the considerable benefits of producing a model of any proposed prototype system. The model should be capable of being exercised with a realistic workload to provide performance indicators and to enable the effects of design decisions to be explored.

The work presented in this thesis is concerned with the architectural modelling of an Asynchronous Bipolar Microprocessor. The prototype processor design is derived from AMULET1 [Furb94a], an asynchronous CMOS version of the ARM RISC microprocessor. Although the AMULET1 architecture was not the first asynchronous microprocessor [Mart89], it is the first to overcome the difficult implementation areas of handling interrupts and exact exceptions, and providing multi-cycle instruction support. The implementation technology is based on a high-performance, differential, current-mode logic family developed by GEC-Plessey Semiconductors. As outlined above, a simulation model is desirable before implementing a prototype system. When, as is the case

with this work, a novel system architecture produced using an unfamiliar design methodology is to be implemented on a new advanced bipolar process, then extensive simulation becomes essential.

There are several objectives of this thesis. The first is to introduce the reader to the issues involved in asynchronous logic design in Chapter 2 and to the specific asynchronous design style used for the project in Chapter 3. The second is to familiarise the reader with the chosen system modelling language in Chapter 4 and the high-performance bipolar technology used to implement the prototype system in Chapter 5. The next objective is to show how the system model components are constructed based on the circuit characteristics of the underlying implementation technology. This is presented in Chapter 6. A further objective, achieved in Chapter 7, is to introduce the ARM architecture and explain the operation of the asynchronous implementation. The final objective is to show how the modelling environment is used to incorporate the design methodology and to demonstrate how the information produced by the model may be used to improve the design of the system in Chapter 8.

The structure of the remainder of this thesis is as follows:

Chapter 2 explains the domination of synchronous design techniques in current electronic circuit synthesis. The problems with synchronous design, which are generating renewed interest in asynchronous design styles, are noted. An introduction to asynchronous logic is given, along with the signalling protocols used, and some of the issues involved in delay modelling are considered.

Chapter 3 gives an introduction to the particular asynchronous design methodology used in the development of the Asynchronous Bipolar microprocessor. Examples of the control circuit elements used are included.

Chapter 4 presents the modelling environment and demonstrates some of the language constructs and the hierarchical structure capabilities. An indication of how time is managed while exercising the model is given.

Chapter 5 introduces the differential bipolar technology employed to implement the prototype system. The circuit operation is explained by considering some gate function examples.

Chapter 6 shows how the architectural models of the bipolar logic gates and functions are developed based on the circuit simulations of the equivalent transistor models of the basic gates. The effects of gate output loading and input drive characteristics are explored.

Chapter 7 outlines the ARM target architecture and the instruction set. The structure of the asynchronous bipolar architecture is then presented with detailed examination of the major functional units, namely the Register Bank, Address Interface, Data Interface and Execution Unit. Simulator output waveforms are included to demonstrate the operation of the units.

Chapter 8 illustrates how the architectural model of the asynchronous ARM was developed in the modelling environment using a hierarchical, modular structure. Some of the features of the modelling language are then elaborated and some examples of the modelling tools that have been constructed are demonstrated. Various executable programs used to validate the architecture and measure performance are presented. An illustration of how the system model is used to gain information regarding the operation of the design and subsequently, how this information is used to suggest system design enhancements is given.

Chapter 9 summarises the current state of the project and draws together the conclusions resulting from this work, discussing the applicability of the Verilog to the architectural modelling of asynchronous systems. Future research areas, continuing on from this work are suggested.

The Appendix contains the complete hierarchical Verilog model of the MDCML Asynchronous ARM including the functional subsystems, asynchronous control elements and standard logic gate primitives.

2. Asynchronous Logic

Computer technology has evolved rapidly over the past few decades and the demand for even higher performance machines seems set to continue as computing solutions to new, more complex and computationally intensive problems emerge.

Synchronous design techniques have dominated the field of digital logic synthesis during this development period. This supremacy has been brought about for several reasons:

- ❑ The concepts required to create a synchronous solution to the production of a logic circuit are easily understood - the designer simply defines the combinatorial logic necessary to perform the required function and then surrounds it with latches which are enabled with a common clock. In a large design, the entire system is then a composition of subsystems communicating by passing data values between the clock-controlled registers.
- ❑ The global clock fulfils two system functions - the clock *transitions* define the successive instants at which the system state changes can occur and the clock *period* is sufficient to account for the logic and wire delays. Since the clock period is specified to be greater than the slowest combinatorial path that could occur during the computation, circuit hazards and feedback problems can generally be ignored [Seit80].
- ❑ By neglecting the effects of *clock skew* - the time difference between the arrival of the global clock signal at different points in the system - the total system state, when considered at the end of the clock period, is assumed to be deterministic and discrete, changing only at the edges of the system clock.

❑ The synchronous design style is well-understood and formalised and is therefore readily accessible to potential digital logic engineers, the preponderance of synchronous circuits is then reinforced when these new engineers become productive.

❑ Also, widely available standard components, which are well-specified and documented, have been positively developed for use in the synchronous style.

❑ Verification of the correct operation of a synchronous design simply involves checking the setup and hold times of the outputs of the combinatorial logic sections of the design to ensure that they meet the requirements of the clocked registers.

❑ CAD tool support has also been developed, in parallel with the synchronous design concepts, which manage much of the timing verification involved.

❑ Testing is also a much easier proposition in a synchronous circuit since many techniques including, for example, Scan Paths and BILBO (Built-In Logic Block Observation) are well-developed.

Recently, however, significant interest has arisen in the field of asynchronous logic design. This interest may be as a consequence of the problems associated with the global clocking strategy becoming more acute, a recognition that the formal techniques for handling asynchronous behaviour and the automatic synthesis potential of asynchronous circuits are now worth exploiting, or that inspiration has been generated by recent publications in the field, most notably the 1988 Turing Award Lecture on *MICROPIPELINES* given by Ivan Sutherland [Suth89].

As VLSI process technologies develop and feature sizes shrink, the global clocking schemes currently employed in synchronous systems are now beginning to experience severe difficulties in the following areas:

❑ Since the clock signal controls the entire system, it must be distributed across the entire chip. This requirement for large scale clock driver circuitry is expensive - in current high performance microprocessors a considerable proportion of the silicon area used and power dissipation required is given over to the global clock logic [Dobb92].

❑ The design effort needed for the clock driver circuitry, and consideration of the effects of clock skew, is non-trivial. It is becoming increasingly difficult to maintain the clock skew within reasonable bounds across all process, temperature and circuit operational speed parameters and may result in the clock period being extended. In current leading-edge synchronous microprocessor designs, a significant proportion of the clock period is used to account for the effects of clock skew.

❑ The circuit modifications required when a relatively small subsection of the system is changed may have ramifications across the entire chip design.

❑ The global clock period must allow for the worst-case logic delay, even though, if the system is not operated in an extreme environment, the worst-case delay may never actually occur. The resulting performance is then reduced as the system is effectively idle during the time between the outputs of the combinatorial logic settling and the arrival time of the (worst-case period) clock.

It has long been recognised by logic circuit designers that asynchronous circuits have a potentially higher performance that synchronous circuits, since an asynchronous circuit exhibits average-case performance (the processing commences as soon as the new input data arrives - the time required to complete the computation execution being dependent on the actual input data values). A synchronous ALU, for example, must be particularly designed to allow for worst-case execution time irrespective of the actual input data values presented to the circuit.

In general, arbitration is required when several sources compete for the same service (or resource), since the service request signals may arrive at the shared resource at any time. In a synchronous system, asynchronous inputs are synchronized to a local clock, allowing metastable effects to be (hopefully) resolved in a limited period. An asynchronous system can wait an arbitrary time for arbitration to occur before making a clear decision. As a result arbitration is inherently more robust and reliable.

However, the asynchronous design framework is unfamiliar to established engineers. The basic ‘building blocks’ of asynchronous logic synthesis need to be developed, since

currently the components are unfamiliar and unoptimised. Also, the circuit size of an asynchronous solution, relative to the equivalent synchronous design, is possibly increased (in part, due to the unoptimised basic components); although this may be offset by the non-trivial requirement for clock-driver circuitry for larger systems.

The existence of circuit races or hazards causes a further complication in an asynchronous design. Fundamental Mode operation[1] must be employed, or various assumptions must be made regarding the relative delays or speeds of the circuit component elements. The testing of asynchronous circuits also causes problems. Sequential circuits are very difficult to test and techniques have not yet been fully developed to test asynchronous combinatorial logic. No high-level method has yet been produced to assist in checking the *liveness* (absence of deadlock) of a design.

Several methodologies have been developed to synthesize asynchronous circuits, some are based on enhancements to Petri nets [Pete81, Moln92], others are compilation-based on high-level languages [Mart90, Brun91] developed from CSP [Hoar85]. Surveys of asynchronous design methodologies and techniques can be found in [Gopa90, Hauc93]. Some of the asynchronous design terminology that may be encountered in the text will now be explained. This relates to the modelling of signal propagation delays and the mechanisms used for communication between asynchronous subsystems.

2.1 Delay Modelling

In a *BOUNDED DELAY* model, it is assumed that the delays in the circuit elements and wires are known (or at least have some upper bound). When input signals are applied to a circuit, then after a particular time interval has elapsed the output signals are known to be valid. Note that this is also the delay model used for synchronous designs.

DELAY-INSENSITIVE circuits use a contrasting model to that used in bounded delay circuits; it is assumed that all signal delays in both elements and wires are unbounded.

1. Fundamental mode operation requires that a circuit achieve a stable internal state after every individual input signal change.

No matter how long the circuit waits, there is no guarantee that an input signal will be received. Circuits designed in this style must include functions to detect when a new input value actually arrives.

The *SPEED-INDEPENDENT* model is a weaker form of the delay-insensitive paradigm, in that it is assumed that the element delays are unbounded but the interconnection wires have zero delay.

2.2 Signalling Protocols

Communication between modules or subsystems in an asynchronous environment is achieved by employing a commonly agreed set of control signals (and some associated operational rules) which are passed between adjacent modules. The method usually involves detecting an ‘event’ on the control signals, eg. a change in the voltage level of the interconnecting wire.

In order to construct asynchronous systems by the composition of individual subsystems, where each performs a specific (and different) function, a general signalling protocol is required. This protocol will operate between the various modules without any regard to the internal processing rates of individual modules, or of the actual signal propagation delays of the communication links. This can be achieved by placing no restrictions on the timing of the signals involved in the communication protocol. Only the *sequence* of the control signal transitions is significant.

The basis for some of the simplest protocols involves the use of two wires connected between adjacent modules: a REQUEST wire and an ACKNOWLEDGE wire.

Asynchronous systems usually employ one of two communication protocols: two-phase (or transition) signalling or four-phase signalling.

2.2.1 Two-Phase Signalling

In this protocol, any transition between the two logic levels, a HIGH to LOW transition or a LOW to HIGH transition, is accorded the same meaning.

A transition may also be referred to as an EVENT, hence the alternative name for this protocol is ‘event signalling’.

Two-phase signalling operates between two modules in the following manner:

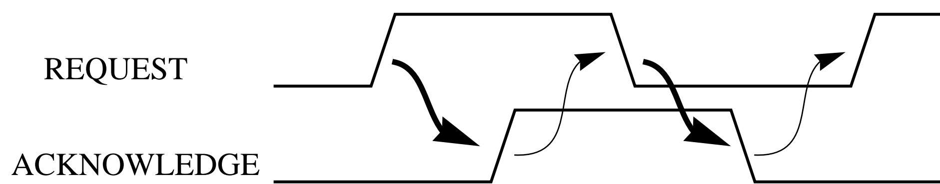


Figure 1 : Two-phase (transition) Signalling.

The sender generates an event (transition) on the REQUEST wire. At some point in time later, the receiver detects the request transition and indicates that it has received the request signal by generating an event on the ACKNOWLEDGE wire. The sender eventually receives the acknowledge event, signifying that the receiver is ready to receive another request.

The arrows on the diagram indicate the constraints on the sequence of events allowed on the control signals used in the protocol. The THICK arrows show the constraint imposed by the receiver: an acknowledge event cannot be generated until a request has arrived. The THIN arrows show the constraint imposed by the sender: the sender cannot generate another request until the previous request has been acknowledged by the receiver. The fact that each of the modules regulates the sequencing of one of the control signals indicates that the correct operation of the inter-module communication path will only occur when both sender and receiver obey the protocol rules.

Because BOTH edges are used in the two-phase scheme - the actual logic LEVEL of a particular control signal does not assume any significance - it provides the capability of

increasing the performance of communication protocols above that of conventional signalling methods, since _every_ change in the signal carries some information content.

Initially, the concepts of transition signalling may be difficult to assimilate into the mindset of the conventional logic designer, since the two-phase circuits must be symmetrical with respect to the high and low logic levels of the control signals.

2.2.2 Four-Phase Signalling

Four-phase (or ‘Return to Zero’) signalling is characterised by the control signals being active when in the HIGH (logic ‘1’) state and then being required to return to the LOW (logic ‘0’) state before subsequently becoming active again.

The protocol could take the following form:

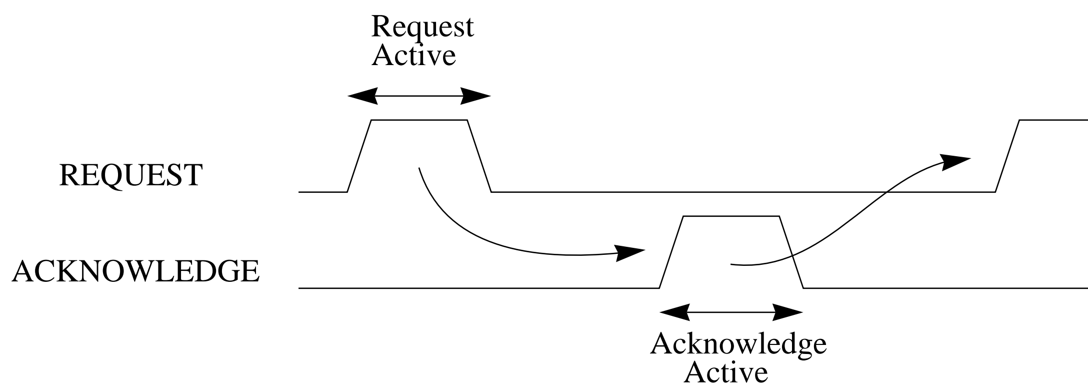


Figure 2 : Four-phase Signalling (incorrect operation).

The sending module raises the REQUEST line to its HIGH (active) state and after a short time interval deactivates the signal by taking it LOW again. The receiver, having detected the request line entering its active state, produces a response by briefly raising the ACKNOWLEDGE line to its HIGH state.

However, the protocol in this form may result in communication failure since, if the sender has a comparatively faster circuit operation than the receiver, the sender may raise then quickly lower the request line to produce a very narrow ‘pulse’ which the receiver may be unable to detect.

Correct protocol operation is enforced by requiring the sender to continue holding the request line in its active (HIGH) state until ‘request reception’ is indicated to the sender by the receiver raising the acknowledge line into its HIGH state. The request line is then deactivated, allowing the receiver to subsequently deactivate the acknowledge line.

The properly functioning four-phase protocol is then:

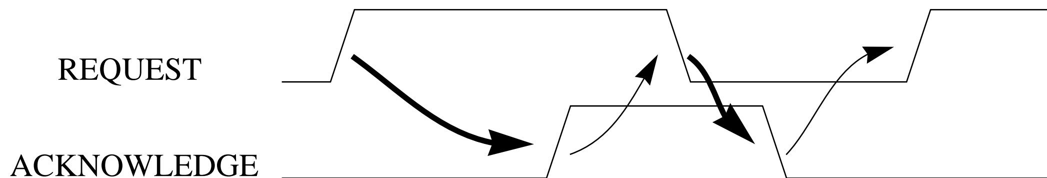


Figure 3 : Four-phase Signalling (correct operation).

Again, each of the modules taking part in the communication imposes constraints on the sequencing of the control signal transitions. The THICK arrows indicate the constraints enforced by the receiver: the acknowledge line can only enter its active state after the request line is activated and can only be deactivated after the request line is deactivated. Similarly, the THIN arrows show the constraints imposed by the sender: the sender must not ‘remove’ the request signal until the receiver acknowledges that it has ‘seen’ the request and a subsequent request must not be generated until the acknowledge has entered its inactive state.

The four phases of the protocol can be observed by noting the four possible combinations of the control signals:

Request LOW, Acknowledge LOW - Inactive

Request HIGH, Acknowledge LOW - Requesting

Request HIGH, Acknowledge HIGH - Acknowledged

Request LOW, Acknowledge HIGH - Request cleared, Acknowledge to clear

Four-phase signalling may be more familiar to current logic designers since each phase of the protocol may easily be determined by examining the logic LEVELS of the control signals.

Also, four-phase signalling is easier to implement because of the widely-available standard components which have been developed to manage logic *levels*.

2.2.3 Data Communications

In addition to the signalling protocols used to indicate control actions, outlined above, a mechanism for passing data values between modules is required.

The simple REQ/ACK scheme can only signal events. In order to transmit data values a method of differentiating between two alternative events (sending a ‘1’ and sending a ‘0’) must be employed. This method could be extended, by using two sets of REQ/ACK pairs, into a four-wire per bit signalling system where each pair is used to communicate a particular bit value: Req0/Ack0 is used to send and acknowledge zeros, Req1/Ack1 is similarly used for ones. In the simplest system, consisting of only four wires, multiple bit values, bytes or words, are sent in bit-serial fashion.

The number of wires required, per bit, may be reduced to three by noting that the two acknowledge wires Ack0 and Ack1 may be combined into one common acknowledge wire, Ack.

This idea of a common acknowledge wire can be used for the communication of multiple bit ‘words’. Two request wires, R0 and R1, are provided for the transmission of each bit (a technique also known as DUAL-RAIL ENCODING) and the common word acknowledge signal is returned only when a transition has occurred on one of the request wires for each of the bits of the word.

2.2.4 Bundled-Data Interface

Although the previous schemes provide a robust communication technique in an environment where signal propagation delays are unpredictable, the cost in terms of number of signal wires needed is high. This is especially the case when the communication is over a relatively long distance. There is also a cost in terms of the signal detection/completion circuitry required when dealing with multiple bit data 'words'.

The *bundled data* interface seeks to significantly reduce the number of signal wires, particularly for large bit-width data values, to just one data wire per bit (as in conventional synchronous 'bus' structures). This set of signal wires is collectively known as a BUNDLE. For each wire, the logic *level* indicates the value. In addition, a request/acknowledge pair of control wires is needed per data word.

Assuming that two-phase (transition) signalling is used on the req/ack control wires, the data values are transmitted in the following manner:

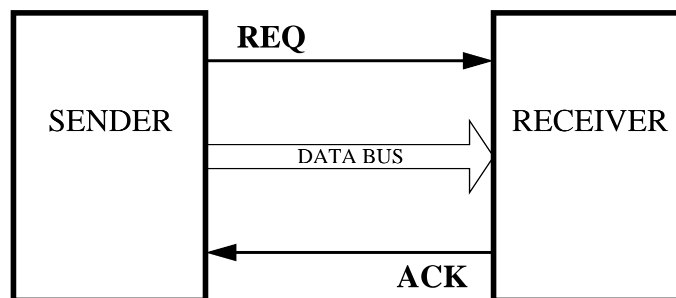


Figure 4 : Bundled Data Interface.

The sender places the n-bit value onto the data wires (bus) and then generates an event on the REQUEST line. At some later time, the receiver will detect the arrival of the request event which will indicate that the data bus is holding the correct transmitted value. The receiver will then latch the data value before generating an acknowledge event back to the sender. The sender is then free to remove the current data value and set up the next value for transmission.

Note that for correct operation, there is an implied assumption that the data value arrives at the receiver before the request event i.e. in the same order as they were generated by the sender. More formally - “The sequence relationships of events in a bundle are the same at the sender and the receiver.” [Suth86]. This is a timing constraint on the use of the bundled-data interface and the logic circuit designer must ensure that this timing relationship is satisfied.

Also note that the sender may not change the data value, once it has generated a request, until it receives an acknowledgment from the receiver. From the point of view of the receiver, the data is only valid from the time of reception of the request event until the acknowledge is generated.

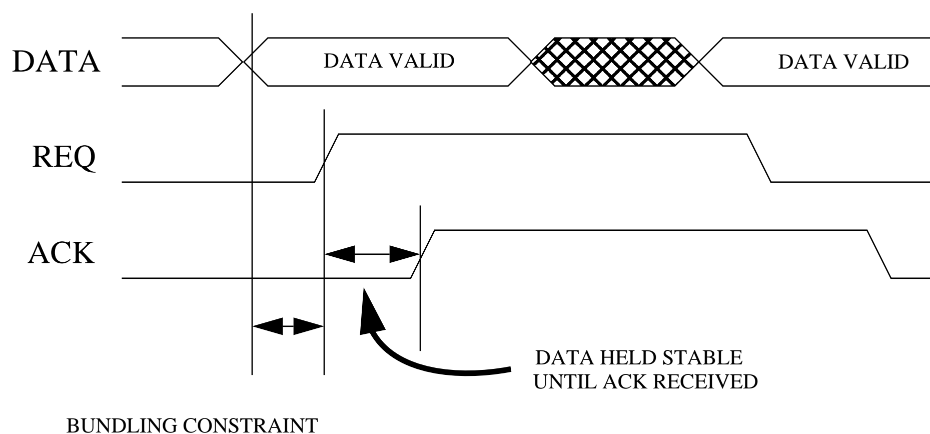


Figure 5 : Data Value Constraints.

3. Micropipelines

Pipelining is used in computer architectures to provide increased processing rates through the use of concurrency [Kogg82]. A large computation is divided into a series of operations, which execute in parallel.

Pipelines may be clocked (*synchronous*) or event-driven (*asynchronous*). In both synchronous and asynchronous pipelines, the *throughput* - the number of data items processed in a given time interval - is limited by the computational rate of the slowest subsystem (module) in the pipeline. However, the *latency* - the time taken for an individual data item to pass through the complete (empty) pipeline - of the synchronous and asynchronous pipelines differs.

For the synchronous case, the latency is calculated to be the number of pipeline stages multiplied by the processing time of the slowest element; the clock period must be specified to accommodate the slowest element, even though all other elements may be capable of sustaining much higher clocking rates.

The latency of an asynchronous pipeline is calculated to be the sum of the processing times of each element. This latency can be significantly less than that of the synchronous case if there is a wide range of processing rates for the component elements.

The lower latency of asynchronous pipelines may be exploited in, for example, processor instruction execution pipelines where the pipeline is frequently flushed when a branch instruction is executed.

Pipelines may also be categorised as ELASTIC or INELASTIC. For an inelastic pipeline, the input and output data rates must match, implying that the total amount of data contained in the pipeline is fixed. When an inelastic pipeline contains no processing elements, i.e. each stage consists of a storage element only, it acts like a simple SHIFT REGISTER. In contrast, the input and output rates of an elastic pipeline, however, may differ momentarily and therefore the amount of contained data is variable. An elastic pipeline without processing elements is a FIFO (First-In, First-Out).

FIFOs provide an important buffering function between systems acting at variable processing rates. The implementation of elastic FIFOs is difficult in a synchronous model: each stage must have a *full/empty* flip-flop, and each stage must be provided with *full/empty* information about the previous and successor stages. A particular stage receives a new data value if, at the appropriate clock transition, the stage is EMPTY and the previous stage was FULL. The stage can then pass on the data value if, at a subsequent clock transition, the next stage is EMPTY. The current stage can then make itself available to receive a new data value by changing its state flip-flop from FULL to EMPTY.

Also, since the clocking rates at the input and output of an elastic pipeline may be different, some form of arbitration and synchronisation will need to be provided between the FIFO and the systems connected to it. An asynchronous implementation removes the requirement for arbitration by allowing the input and output processes of each stage of the FIFO to operate at their own pace.

In 1988, at the Turing Award Lecture, Ivan Sutherland put forward a modular approach to the design of computer systems using asynchronous logic. His idea was based on the use of simple data processing pipelines whose stages operate asynchronously. He termed these 'MICROPIPELINES' [Suth89]. A micropipeline is an elastic, bounded-delay, event-driven system using transition signalling and the bundled-data interface. A micropipeline without processing, a simple elastic FIFO, can be constructed from a basic component known as an Event Register (see Section 3.2.1).

One of the benefits of micropipelines is that the registers, used to hold the data values as they flow through the pipeline, can be used to filter out hazards. This is achieved, in a similar manner to that used in synchronous designs (where the clock period is sufficiently long to account for hazards), by locally delaying the request output signal until all data values are stable. Also, any combinational logic structures can be used 'between' the pipeline registers including existing circuits used in synchronous designs.

As discussed in Chapter 2, the bundled-data interface is useful for data communications since it reduces the number of data wires required to transmit a value, particularly for large numbers of bits. In implementing an asynchronous 32-bit processor, for example, the inter-module completion-detection circuitry required if the data was transmitted using the '2 wires per bit' protocols would be prohibitively large.

Micropipelines offer the opportunity to construct complex systems by the hierarchical composition of simpler modules. The two-phase signalling protocol allows modules of widely-differing performance to be easily integrated into a complete, correctly functioning, system. The data-driven execution rates of the individual asynchronous modules allow the benefits of average-case performance. The micropipeline approach also provides the facility to replace a particular module with one of a higher performance without impacting on the correct operation of the total system (as would be the likely case with a synchronous global clocking scheme).

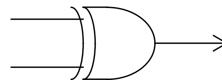
In the context of VLSI technology, the design cost of large systems both in terms of time and effort is beginning to outweigh the combined fabrication and production costs of the final integrated circuits. Since an 'ad hoc' design style is impractical for large scale circuits, micropipelines provide a basis for an asynchronous design methodology for the construction of such systems. Pre-synthesised modular solutions to standard problems, packaged in an asynchronous design library, can then be interconnected using the transition signalling protocol.

The circuit designer simply ensures that each module conforms to the interface protocol and need not be fully conversant with the internal intricacies of the asynchronous ‘cells’. The inefficiencies of this ‘standard module’ approach may be negligible when compared to the extra design cost of a full custom approach.

3.1 Control Circuit Elements

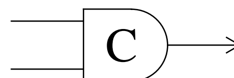
The transition signalling control circuits used to coordinate the activities of micropipelines may be constructed from a standard set of ‘event logic’ modules [Suth86]. Some of the more widely-used modules are presented below.

3.1.1 XOR (Merge)



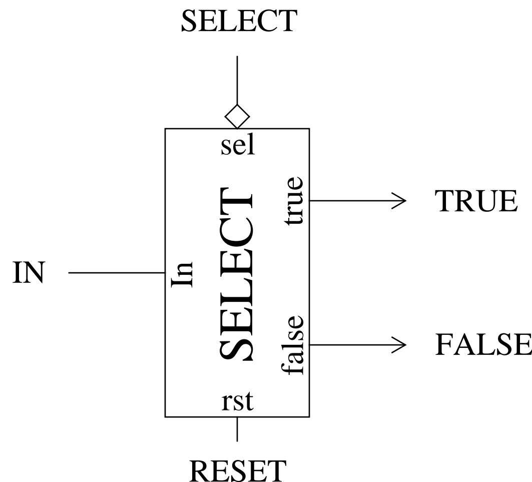
The Exclusive-OR (or non-equivalence) gate provides an ‘OR’ function for transition signals. An output transition (event) occurs in response to a transition arriving at any of the inputs. This module is also known as a MERGE element.

3.1.2 Muller-C (Join)



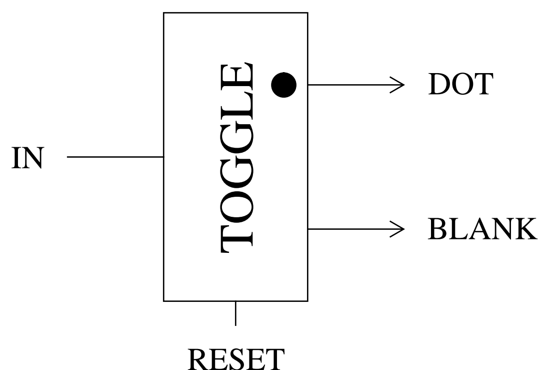
The Muller-C element serves as an ‘AND’ function for events. A transition occurs on its output only after a transition has occurred on each of its inputs. In logic level terms, when the input levels match, the output assumes the same logic level as the inputs, otherwise the output holds its previous level. A reset input may be added to force the output to a defined initial state. The standard AND logic symbol with a large ‘C’ inside is used to represent the Muller-C element, which is also known as a JOIN or RENDEZVOUS element.

3.1.3 Select



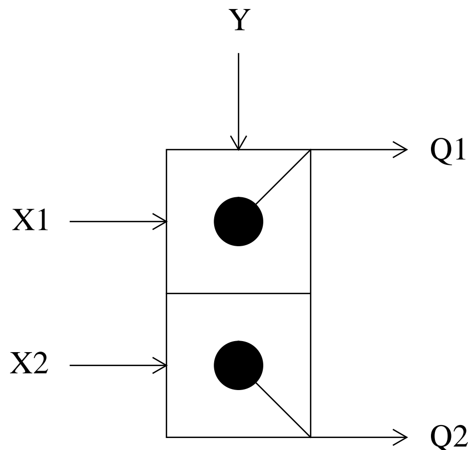
The Select element ‘steers’ an input transition to one of the two outputs depending on the Boolean value of a second, ‘select’, input. The select Boolean value must be valid before the input transition occurs. This is, effectively, a bundling constraint on the IN (event) input. The module is NOT delay-insensitive because of this requirement. Furthermore, while the Select element is essentially an event-triggered device, the logic level of the select input is significant.

3.1.4 Toggle



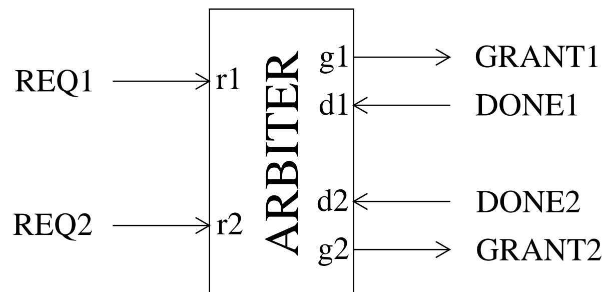
In a similar manner to the Select element above, an input transition of the Toggle element is steered to one of the two outputs. However, the output event is produced alternately on the two outputs in response to an input transition. Following a Reset signal, the first output to receive an event in response to an input event is marked with a heavy dot (see diagram), the outputs are then known as Dot and Blank (i.e. no dot).

3.1.5 Decision-Wait



A Decision-Wait element has two sets of input signals and produces an output event when one event in each input 'set' has been received. For example, an event on input Y and an event on either X1 or X2 will produce an output event on either Q1 or Q2 respectively. Note that for correct operation, only one input event can be received on an X input (X1 or X2) for each event received on the Y input before the appropriate output transition occurs.

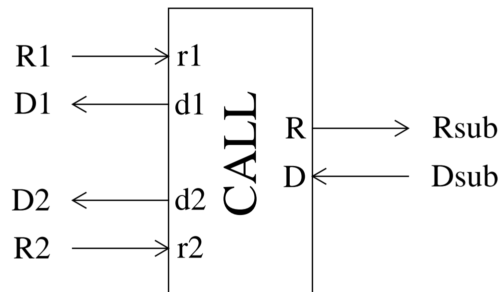
3.1.6 Arbiter



An Arbiter is used to guarantee mutually exclusive access to a shared resource for two competing independent requests. The arbiter chooses only one of the active input requests and allows only the corresponding output grant signal event to occur. When the arbiter is already in use by a requester, a second requester is inhibited until the "request done" acknowledge event is received (DONE1 or DONE2, depending on the currently active requester) indicating that the active requester is releasing control of the arbiter. The arbiter will then issue a grant signal (event) to the second requester.

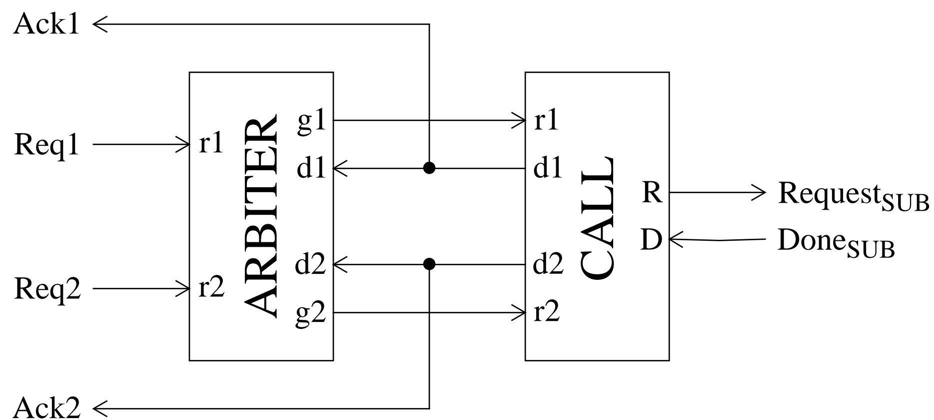
Although the input requests can occur at any time, even simultaneously, the output grant signals are guaranteed to be mutually exclusive or serialized.

3.1.7 Call

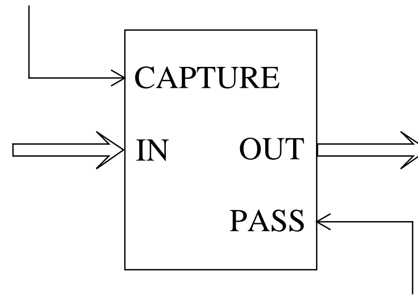


The Call element provides the functional equivalent of a “subroutine call”. A request event for access to a common hardware function is received on one of the two request inputs, R1 or R2, which will subsequently generate a ‘subroutine’ request event on the Rsub output. When the subroutine function has completed, indicated by the arrival of an event on the Dsub (subroutine done) input, the Call element generates an output event on the appropriate ‘request done’ output (D1 or D2, depending on the active requester).

For correct operation, the full Request / Subroutine Request / Subroutine Done / Done cycle must complete before the next Request occurs and therefore the two input request signals, R1 and R2, must be mutually exclusive. For a circuit topology where R1 and R2 cannot be guaranteed to be mutually exclusive, the input requests may be routed to the Call element via an Arbiter.



3.1.8 Capture-Pass Latch



The Capture-Pass latch is a storage element for use in an event-controlled system; unlike a traditional level-sensitive latch in which the *high* and *low* states of the clock/ enable signal indicate a different function, the event-controlled latch must provide equivalent responses to rising and falling transitions.

When the Capture and Pass inputs are in the same state (either both *high* or both *low*), the latch is in the **PASS** state: the output of the latch follows any change in the input value. When the Capture 'event' occurs, the latch will become insensitive to changes in the input data and will **CAPTURE** (store) the current input value, resulting in the output value being held stable. After a subsequent Pass 'event', the element will become transparent and the output will again follow the input.

For the Capture-Pass latch to operate correctly, the Capture and Pass events must alternate.

3.2 Control Circuit Examples

3.2.1 Event Register

As mentioned previously, an asynchronous FIFO can be constructed from a basic component known as an Event Register.

An Event Register uses a two-phase signalling protocol on its input and output control circuits and incorporates an event-controlled storage element for the associated bundled-data value.

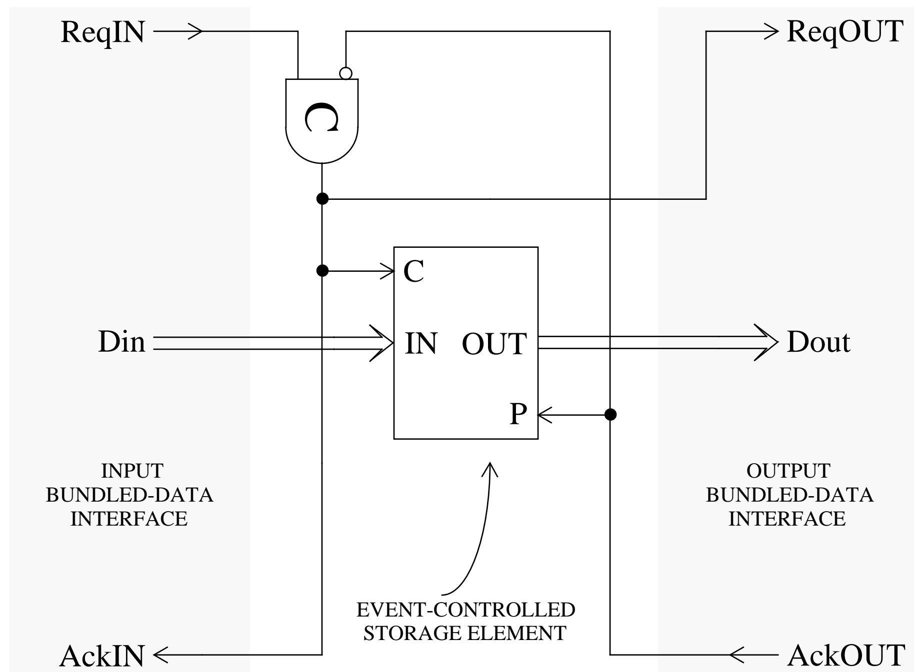


Figure 6 : Event Register.

Event Registers with 32-bit data values are used extensively throughout the Asynchronous ARM design.

The operation of the Event Register is as follows:

i) Initially, assume all signals are *LOW* and the Capture-Pass latch is in the *PASS* (transparent) state.

ii) An input data value is supplied followed by the arrival of a ReqIN event at the Muller-C and, because of the input inversion of the *LOW* state of the other input, an output event is generated from the Muller-C.

iii) The Muller-C output event causes the Capture-Pass latch to enter the **CAPTURE** state: it latches the data value presented on its input.

iv) Once the Capture-Pass latch has captured the data, an AckIN event is sent to the ‘previous’ stage (the previous stage can now prepare a new

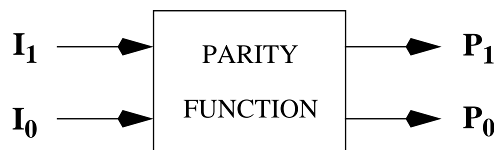
input data value) and a ReqOUT event is sent to the 'next' stage of the pipeline.

Note that the Event Register will only capture the data in response to a ReqIN event if the stage is currently *EMPTY*. That is, if an acknowledge from the output stage is not pending from a previously generated ReqOUT event to the next stage.

Also, the Capture-Pass latch must be in the PASS state (Dout valid) before a Capture event occurs and, since the ReqOUT event is equivalent to the Capture event, the Dout data value must be valid before the ReqOUT event. The output interface of the Event Register therefore obeys the data bundling constraint.

3.2.2 Design Example: PARITY FUNCTION

A dual-rail encoded parity function using a transition signalling protocol was presented to the IFIP Working Conference on Asynchronous Design Methodologies (April 1993, Manchester) by Charles Molnar and this will be used as a design example:



The circuit will receive an input signal as an event on one of the two input wires, I_0 or I_1, depending on whether a '1' or a '0' is indicated. The circuit must then provide an output event on one of the two output wires, P0 or P1, to indicate the *cumulative* parity of all of the inputs received up to that point.

A general, high-level, formal method of synthesizing Micropipeline control circuits does not, as yet, exist. A pragmatic approach must therefore be taken to derive a design for the parity function circuit using the Micropipeline control blocks outlined previously in Section 3.1.

Assume that after a global reset, all the interface signals (inputs and outputs) of the parity function are *LOW*.

It can be noted that the accumulated parity of the received inputs is actually given by the logic level of the I_1 input. If I_1 is *HIGH*, the accumulated parity is ‘1’, if I_1 is *LOW*, the parity is ‘0’.

Events occurring on the I_1 input cause alternating output parity signals to occur, since an I_1 event indicates the arrival of a ‘1’ and this will cause the accumulated parity to change. A **TOGGLE** element can be used to alternate the parity output events when the I_1 input event indicates another ‘1’ has been received.

Events occurring on I_0 (indicating a ‘0’ has arrived) cause an output event on P_1 or P_0 depending on the current accumulated parity value. That is, the output to be activated is indicated by the logic level of the I_1 input signal. A **SELECT** block can be employed to ‘steer’ the I_0 input event to the appropriate parity function output based on the logic level of I_1.

Two **XOR** elements are used to merge each of the separate sources of the P_1 and P_0 output events (from the Toggle and Select blocks) onto the actual parity function outputs. The Micropipeline control block implementation[1] is shown in Figure 7.

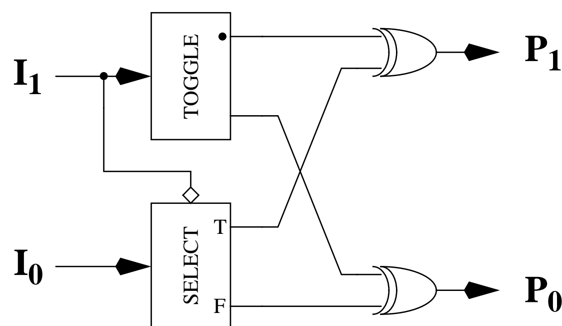


Figure 7 : Micropipeline Control Block Implementation of Parity Function.

1. Since Micropipeline control circuits are usually concerned with the control of datapath elements using the bundled-data convention, this example circuit is not typical of those found in an asynchronous Micropipelined microprocessor.

4. Verilog HDL

4.1 Introduction to HDLs

When viewed at its lowest level, a digital system, particularly in the context of a VLSI implementation, may consist of several hundreds of thousands of primitive components. These components may be transistors or simple logic gates. At a higher level, these elements may be logically grouped into functional units such as Arithmetic Logic Units (ALUs), cache memories and Floating Point Units (FPUs) [Thom92].

Hardware Description Languages (HDLs) have been developed to assist the design process of such systems in managing the complexity involved in the synthesis of complex digital systems [Hart87]. The system may contain a large number of elements and a wide range of logical and physical implementation abstractions, in order to give a total overview of the system.

Initially, a conceptual idea of the required logical system is coupled with a set of constraints (relating to performance, power requirements, circuit size etc.) that the implemented system must meet and a set of primitive components from which to construct the system. The creative design process is an iterative operation of either manual composition or automatic synthesis of alternative solutions, which are then compared against the given system constraints. Normally, the design is partitioned into smaller sub-units, in the classical engineering technique of “divide and conquer” (or top-down design), and each sub-unit is then further divided until the complete system is specified in terms of the known primitive components [Brow91].

4.2 Introduction to Verilog

The Verilog[1] Hardware Description Language [Veri92] describes a digital logic system as a collection of textual-based models that define the functionality of the component sub-units and connections to those sub-units. The language accommodates a wide range of levels of abstraction:

ALGORITHMIC - the component's operation is expressed in high-level (program-like) language constructs.

REGISTER TRANSFER LEVEL (RTL) - the flow of data between registers is described.

GATE LEVEL - the system is defined in terms of logic gate primitives and their interconnections.

SWITCH LEVEL - for low-level design, particularly for MOS implementation, the system may be described in terms of transistors and storage nodes.

The language supports the early conceptual stages of design with its behavioural levels of abstraction (algorithmic and RTL), and the later implementation stages with its structural levels of abstraction (gate and switch levels).

During the design process, behavioural and structural constructs can be mixed as each of the sub-systems is designed. Hierarchical constructs are also provided to allow the system designer to control the complexity of the description.

4.3 Modules

Verilog describes a digital system in the form of a set of MODULES. The logical structure of each module is expressed either in logic gate (or MOS primitives) terms or as a behavioural representation.

---

1. Verilog is a trademark of Cadence Design Systems, Inc.

A module definition includes declarations of the external interface presented to other modules and any internal state used by the module. The external interface is defined in terms of PORTS, which are specified in parentheses after the module name. Ports may be declared to be INPUTS, OUTPUTS or bidirectional INOUTS. A module body contains either behavioural statements which specify the functionality of the module, or statements which create instances of other user-defined modules or logic gate primitives. By allowing module definitions to instantiate other modules, a hierarchical description of the system can be specified. The use of a hierarchical modular approach accommodates the “bottom-up” and “top-down” design styles.

4.4 Structural Modelling

A structural representation of a functional unit is achieved using gate and/or switch level modelling. A set of 26 standard gate-level primitives are incorporated and these can be extended by employing user-defined primitives. This provides a compact and efficient way of describing an arbitrary block of logic.

The Verilog HDL facilitates the accurate modelling of signal contention, bidirectional pass gates, resistive MOS devices, dynamic MOS, charge sharing and other technology dependent network configurations by allowing net signal values to have a wide range of unknown values and different levels of drive strengths.

A declaration begins with the gatetype keyword specifying the required gate or switch primitive. Gatetype keywords include: **and, or, not, buf, nmos, pmos, pullup** etc. Gate and switch instances include an optional instance name and a required terminal connection list.

The propagation delay from input to output through a logic gate or switch primitive may be specified in a declaration. The drive strengths on the output terminals of a gate declaration instance may also be defined.

'Nets' are a fundamental data type of the language and are used to model an electrical connection. Except for the **trireg** net, which models a wire as a capacitor that holds electrical charge, nets do not store signal values. Nets only transmit values that are driven on them by structural elements (gate outputs or assign statements) or behavioural models (registers).

4.4.1 Design Example: RS Flip-Flop

An RS flip-flop consists of two inputs, SET and RESET, and (normally) two outputs, Q and $\overline{Q}$. For the purposes of this design example, the $\overline{Q}$ signal will not be generated as a module output. All the signals, inputs and output, will be active *HIGH* i.e. positive logic.

When the SET input is asserted (HIGH), the Q output signal goes HIGH and remains HIGH even when the SET input is deactivated. When the RESET input is asserted, the Q output goes LOW and again stays LOW when the RESET signal is deasserted. A conflict will occur if the SET and RESET inputs are both HIGH simultaneously. The logic circuit designer should ensure that this situation never arises.

A circuit implementation consists of two cross-coupled NOR gates [Mano84]:

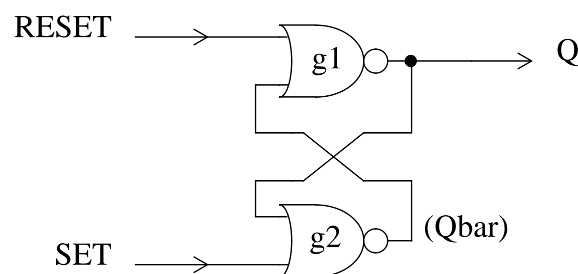


Figure 8 : Possible Implementation of RS Flip-Flop.

An example module of a structural (gate-level) representation of a RS flip-flop is given overleaf. Each module definition begins with the **module** keyword and is terminated by

the **endmodule** statement. The first line of the definition specifies the module name and the names of its ports.

```
module RS_FF(set, reset, Q);          (1)
input set, reset;                     (2)
output Q,                             (3)
wire Qbar;                            (4)

nor #10 g1(Q, reset, Qbar);           (5)
nor #10 g2(Qbar, set, Q);             (6)
endmodule                             (7)
```

In lines 2 and 3 in the example, the type of each port is specified: set and reset are input ports, Q is an output port. The module's logic gate primitive components are defined in lines 5 and 6. The first word in the line indicates the component type-name - in this case, **nor** gates. The #10 indicates that the propagation delay of the gate from input to output is 10 time units. The **nor** gates are then instantiated by giving each one an instantiation name (g1 and g2) and specifying the gate connections. The output is specified first, followed by any number of inputs (in this example two). There is a net (or wire) which is internal to the module, i.e. it is not an input or output, which connects the output of g2 to an input of g1. This internal net is declared and named in line 4.

4.5 Behavioural Modelling

When a system is modelled as a structural, gate-level representation, very little translation effort is required to convert the HDL model into a correctly functioning physical implementation. However, in many cases the circuit engineer requires the opportunity to derive many design alternatives and consider the merits of each design solution. Behavioural modelling facilitates the architectural refinement of a design. It allows the higher-level functional aspects of the prototype system to be easily evaluated in isolation, without regard to the final implementation of the proposed circuits [Russ89].

The syntax of the Verilog behavioural language is very similar to the high-level programming language 'C' [Kern88]. It contains a number of procedural constructs which

include the familiar **if-then-else** conditional execution construct, the conditional assignment (**?:**) operator and the multi-way branch **case** statement. Four different statements are provided for iterative sequential behaviour: the **for, while, repeat** and **forever** loops. A full range of arithmetic, logical, bit-wise and reduction operators are also incorporated.

4.5.1 Compound Statements

Two or more statements may be grouped together by means of a *block* statement so that, syntactically, they act like a single statement. In a SEQUENTIAL block, which is delimited by the keywords **begin** and **end**, the statements execute in sequence. Control passes out of the block when the last statement executes. The delay values used in the assignment statements are relative to the execution time of the previous statement:

```
begin
    #10 a = 1;
    #5  b = 0;
    #10 c = a;
end
```

In the example, register **a** is assigned the value 1 ten time units after the execution of the block statement commences. A further five time units later, i.e. fifteen time units from the start of the block statement, register **b** is assigned the value 0. Register **c** is then assigned the value of **a** (now equal to 1) a further ten time units later.

The keywords **fork** and **join** surround a CONCURRENT block statement in which the individual statements execute in parallel. Delay values in assignment statements are relative to the simulation time on entry to the block and control passes out of the block when all of the statements have executed:

```
fork
    #10 a = 1;
    #15 b = 0;
    #25 c = a;
join
```

To achieve the equivalent effect to the sequential block statement, the assignment to register **b** in the second line must have a delay value of fifteen time units, since the delay is relative to the start of the block (not relative to the previous statement, as in the sequential block example). In a similar manner, the assignment to register **c** has a delay value of twenty-five time units.

4.5.2 Process Control

The essence of a Verilog behavioural model is a PROCESS, which can be thought of as an independent flow of activity. The dynamic behaviour of a digital system is then a set of independent, inter-communicating processes. The basic Verilog control construct for describing a process is the **always** statement:

```
always
    <statement>         // Continually repeats
```

The always construct continually repeats the statement following, which may be a block statement (outlined earlier). All of the functionality of a module should be specified within an **always** statement.

A further Verilog control construct, called the **initial** statement, describes a process that is executed only once - it provides a means of initialising signals and internal module state variables:

```
initial
    begin
        busy = `false;     // Initialise values
        out = 0;
    end
```

During simulation of a model, all of the activity flows defined by the **initial** and **always** statements start together at simulation time zero.

4.5.3 Timing Control

Two types of explicit timing control are provided in Verilog to regulate when procedural statements are to occur in simulation time. The first type is a *delay* control in which

a value expression specifies the time duration between the activity flow reaching a particular statement and the simulation time at which the statement actually executes. The second type of timing control is the *event* expression, which allows the execution of statements in a particular procedure to wait for the occurrence of some simulation event. The awaited simulation event will be generated by some other, concurrently-executing, procedure. A simulation event can be either the change of a value on a net, or in a register (an IMPLICIT event), or the occurrence of an explicitly named event that is triggered from other procedures (an EXPLICIT event). In many cases, the event control is the positive or negative edge of a clock signal.

Simulation time can only advance by one of the following three methods:

- A *delay* control, which is introduced by the number symbol (#):

  eg. **#100** out = ~in;

  After 100 time units, the output is defined to be the inverse of the input signal.

- An *event* control, which is introduced by the at symbol (@):

  eg. **always** @(negedge clock)
  out = ~in;

  At every clock transition from HIGH to LOW, the output becomes the inverse of the input.

- The **wait** statement, which operates like a combination of a **while** loop and an *event* control:

  eg. **wait** (reset)
  out = 0;

  Suspend the process until the ‘reset’ signal is HIGH. When the reset signal does eventually go HIGH, the output signal is forced to zero.

4.5.4 Design Example: Behavioural Representation

```
module RS_FF(set, reset, Q);                                   (1)
input set, reset;                                              (2)
output Q;                                                      (3)
reg Q;                                                         (4)

initial                                                        (5)
    Q = 0;                                                     (6)

always @ (set or reset)                                        (7)
    case ({set,reset})                                         (8)
        2'b10:  #10 Q = 1;                                     (9)
        2'b01:  #10 Q = 0;                                     (10)
        2'b11:  begin                                          (11)
                    $display("RS_FF: SET and RESET active");   (12)
                    #10 Q = x;                                 (13)
                end                                            (14)
    endcase                                                    (15)
endmodule                                                      (16)
```

Again the module definition is enclosed in the **module** and **endmodule** keywords and, as in the structural representation, the ports and port types are declared in lines 2 and 3.

Line 4 declares a register with the same name as the output, Q, which will (implicitly) drive the output. Any value assigned to Q will be stored in the register and any value held in the register will be propagated to the output port. Registers are an abstraction of storage devices found in digital systems. Single-bit registers (like Q in the example) are termed scalar; multiple-bit registers are termed vector (eg. addr[31:0] is a 32 bit register).

The **initial** statement in line 5 is executed only once at the commencement of the simulation. This provides a mechanism for initialising the output value.

The **always** statement in line 7 is used to provide the dynamic functionality of the module. **always** @ (set or reset) indicates that the following statements should be executed whenever there is a change to one of the specified signals, i.e. the inputs. The **case** statement on the following line provides a decision capability based on the values of the SET and RESET inputs. Line 9 means that Q will be set HIGH, if the values of the SET and

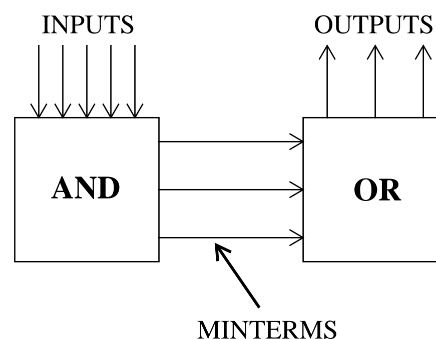
RESET inputs, when concatenated together ({set, reset}), match the 2-bit Boolean value 10 (2'b10). Basically, if the SET input is HIGH and the RESET input is LOW, then the output (register) Q will be set. Similarly, in line 10, if the SET input is LOW and the RESET input is HIGH, the output is driven LOW (reset).

Lines 11 to 14 indicate an important feature of the Verilog behavioural language, namely the ability to report diagnostic messages to the logic circuit designer while the simulation is running. As mentioned in the introduction to the design example, the SET and RESET inputs should never be active simultaneously. If this condition is detected, at line 11, the compound statements (enclosed in the **begin** and **end** keywords) on lines 12 and 13 are executed. An appropriate error message is displayed and the output value is set to undefined (x).

4.5.5 Programmable Logic Arrays

Verilog allows the modelling of both a synchronous and an asynchronous programmable logic array (PLA). The synchronous form allows the designer to control the simulation time at which the array will evaluate the inputs and update the outputs. For the asynchronous type, evaluation is performed automatically whenever an input term changes value.

PLAs are modelled using 2 orthogonal planes:



The logic equations of the separate planes are defined by loading individual data files containing the associated bit patterns.

4.6 Verilog Simulator

The Verilog description of the system may be simulated using a digital logic simulator. This is a software tool that allows many design process tasks to be carried out without the various costs involved in constructing a hardware prototype. These design activities may include [Russ85]:

- Functional Verification.
- Identification of design errors.
- Determination of the feasibility of new design ideas.
- Timing Analysis
- Evaluation of several approaches to a design problem.

The simulator exercises the system model by applying external input signal stimuli. Any generated register or gate output signal changes are then propagated to other gate and module inputs. The main characteristic of the simulator is the ability to manage the concept of time; causing the changed signal values to appear at some specified time in the future. These predicted signal changes are typically stored in a time-ordered event queue.

The RS flip-flop behavioural representation given in Section 4.5.4 is now used as an example to demonstrate the Verilog simulator operation.

In the top-level simulation test file (shown overleaf) the flip-flop module is instantiated (**RS_FF**) with the instantiation name, *f1*. The input signal names are *set* and *reset*, and the output signal name is *Q*. In the first **initial** statement block, the input stimulus sequence is specified. In the second **initial** statement block, the required waveform output display is configured using the **$gr_waves()** system task.

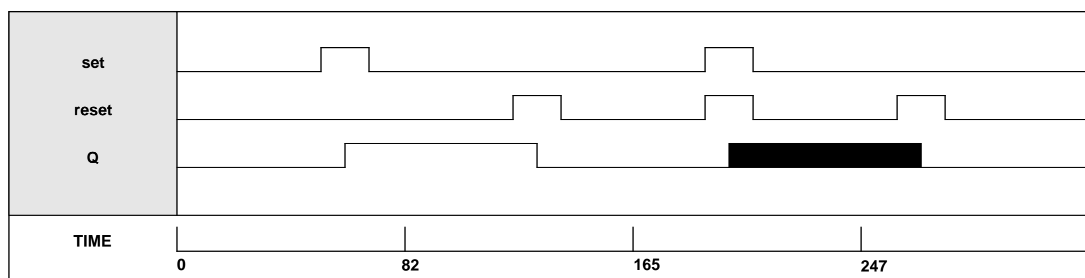
```
`timescale 1ps /1ps
module test();
reg  set, reset;
wire Q;

RS_FF    f1    (set, reset, Q);

initial
begin
    set = 0; reset = 0;

    #50 set = 1;   #20 set = 0;
    #50 reset = 1; #20 reset = 0;

    #50 set = 1; reset = 1;
    #20 set = 0; reset = 0;

    #50 reset = 1; #20 reset = 0;
end

initial begin
$gr_waves( "set",   set
          ,"reset", reset
          ,"Q",     Q     );
    $freeze_waves; #340 $stop;
    $unfreeze_waves;
    $ps_waves("waves.ps", "RS_FF simulation example", 0, 330);
    #1 $finish;
end

endmodule
```

The console output text generated during the simulation execution is given below. Note the warning message displayed when the set and reset signals are simultaneously active:

```
VERILOG-XL 1.7   Jan 20, 1995  09:27:16

Compiling source file "test.v"
GRAPHICS  4.2.2 Thu May 27 23:28:23 PDT 1993 (cds2082)
Highest level modules:
test

RS_FF: SET and RESET active @ time=190

L29 "test.v": $stop at simulation time 340
Type ? for help
C1 > .
L32 "test.v": $finish at simulation time 341
114 simulation events
End of VERILOG-XL 1.7   Jan 20, 1995  09:27:37
```

The graphical display waveforms can also be directed (using the **$ps_waves()** system task) to a postscript file, which is shown below:

Header: RS_FF simulation example		
User: Robert Kelly		
Date: Dec 7,1994 09:52:46	Time Scale From: 0 To: 330	Page: 1 of 1



5. Multi-Level Differential Current Mode Logic

5.1 Introduction to Logic Families

Integrated circuit technology has developed dramatically over the past few decades, both in terms of gate switching speed and sophisticated circuit design, as a result of fabrication process enhancements and shrinking minimum geometries.

The nature of the semiconductor product market tends to segment customers into two groups: performance-oriented users who seek leading-edge performance technology at virtually any cost, and cost-sensitive users who need the best performance available at a given price. Since semiconductor economies depend heavily on a volume market, it is the more numerous cost-sensitive users who tend to drive the development of mainstream semiconductor technology [John91].

Early integrated transistors were **bipolar**, since these were much easier to fabricate. This fact led to the market success of bipolar transistor logic families (DTL, RTL through to TTL) during the early years of IC development. Eventually, the development of the planar process led to the introduction of **MOS** logic families. Initially, because of the more sophisticated processing requirements of CMOS, NMOS logic dominated. However, as chip sizes increased, power consumption problems emerged and the additional complexity in producing CMOS (the lowest power MOS technology) circuits was justified. CMOS technology has now advanced to become the dominant VLSI technology [West89].

When considering the merits of the various logic families several characteristics are examined:

Transistor switching speed - which translates into logic gate delay.

Noise immunity - a measure of the circuit's resilience to EMI.

Silicon layout size - the degree of integration possible on a given chip size.

Power dissipation - specialist techniques are required for high power circuits.

Fan-Out - the drive capability of the logic gates.

Except for the inability to operate at very high switching speeds, CMOS performs very well when judged against these criteria and as a result currently holds an unassailable advantage in the low and medium frequency ranges of the digital logic market.

At low frequencies, CMOS dissipates considerably less power than bipolar circuits because of its virtually zero *static* power consumption brought about by its low leakage current.

However, as the operating frequency rises, the *dynamic* power dissipation of CMOS becomes the dominant factor up to a point where bipolar technologies actually dissipate less power. The power/speed trade-off point between bipolar and CMOS logic families was claimed to be around 50MHz in 1988 [GPS88]. However, with the continuing enhancements of process technologies (particularly with regard to CMOS) the trade-off figure may currently be higher.

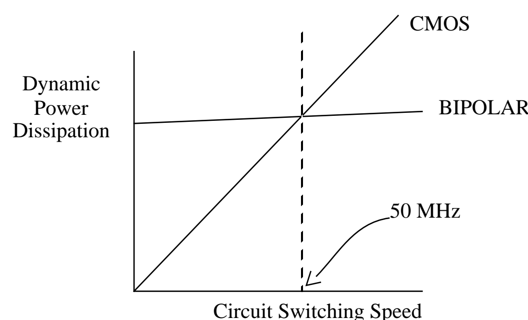


Figure 9 : Dynamic Power Dissipation.

Emitter-Coupled Logic (ECL) operates by "steering current" through a differential ("long-tailed") pair of switching bipolar transistors which are coupled through an emitter resistor. ECL is an extremely fast logic family since it is non-saturating and keeps the logic signal swings relatively small (around 0.8V).

Differential logic is an enhancement of ECL which still uses the long-tailed pair of switching transistors to steer the gate current to one of the two complementary outputs. However, the input signal and its complement are used as the inputs to the switching transistors.

The noise immunity of the gate is increased by the use of the signal and its inverse as inputs, since any noise is experienced as a common-mode signal. The differential amplifier with complementary inputs possesses a high Common-Mode Rejection Ratio (CMRR). The increased noise immunity of differential logic allows much lower voltage swings to be used resulting in a faster gate switching speed (for the same gate current).

5.2 Multi-Level Differential Current Mode Logic

Differential logic (unlike standard ECL or CMOS) can be stacked into a switching "tree" configuration and as a result complex logic functions can be packed into a single gate.

GPS (GEC Plessey Semiconductors) have combined a stacked differential switching tree arrangement with a fabrication process based on Trench-Isolated Bipolar Silicon Technology [Depe89] to produce a logic family known as Multi-Level Differential Current Mode Logic (MDCML) (FAB5 variant).

MDCML has up to 3 levels in the circuit switching tree. This has been chosen as the best compromise between the higher functionality of increasing the number of switching levels and the penalty paid in terms of increased silicon area, the requirement for (voltage) level shifters to transpose signals between levels and the increased power sup-

ply voltage needed to incorporate the many switching levels. GPS estimate that up to 25% of area and 40% of current, in the worst case (i.e. random logic), may be required for level shifting [GPS88].

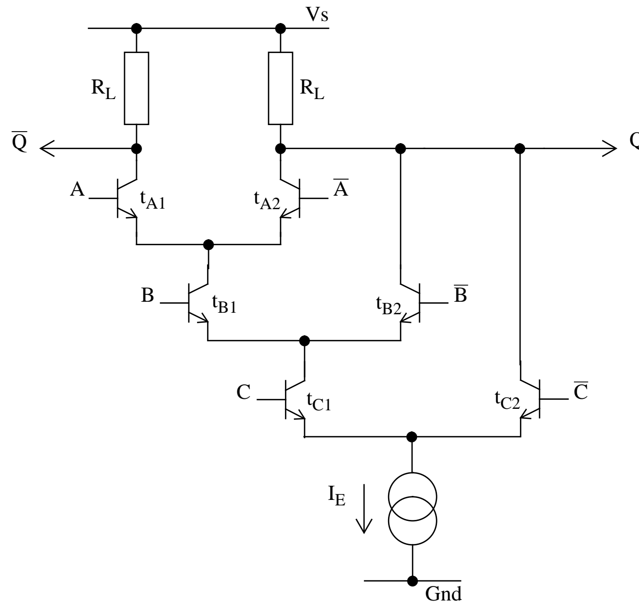


Figure 10 : MDCML 3-Input AND Gate.

A 3-input AND gate structure is shown in Figure 10. There are 3 distinct transistor switching levels. By convention these are known as: LEVEL 3 at the top (inputs A and $\overline{A}$), LEVEL 2 in the middle (inputs B and $\overline{B}$) and LEVEL 1 at the bottom (inputs C and $\overline{C}$). The voltage difference between the switching levels is defined to be one V_{BE} drop, to ensure that the transistors do not saturate, this also simplifies the level shifting circuitry.

The operation of the MDCML 3-input AND gate is as follows:

Assuming all differential input signals are at logic 1, then input A is HIGH and input $\overline{A}$ is LOW; similarly, inputs B and C are HIGH and inputs $\overline{B}$ and $\overline{C}$ are LOW. Transistors t_{A1}, t_{B1} and t_{C1} are ON and transistors t_{A2}, t_{B2} and t_{C2} are OFF. The emitter current flows through t_{A1}, t_{B1} and t_{C1} and causes a voltage drop across the load resistor R_L connected to the collector of t_{A1}. As a result, $\overline{Q}$ is pulled LOW and since no current flows through t_{A2}, t_{B2} or t_{C2}, Q is HIGH.

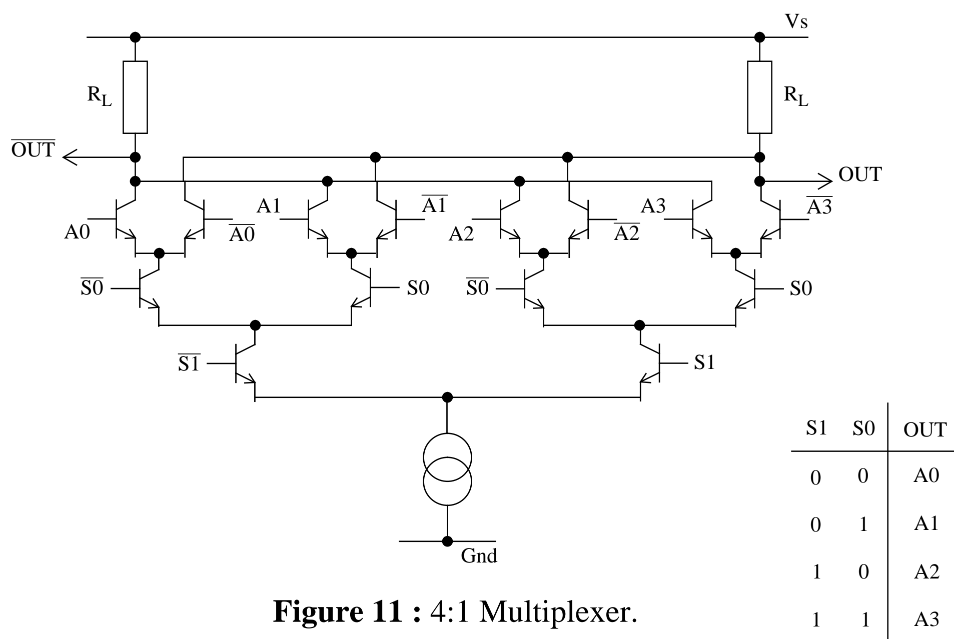
If differential input signal B is then driven to a logic 0, transistor t_{B2} will turn ON (and t_{B1} will turn OFF) causing the emitter current to flow through t_{B2} and t_{C1}, resulting in output Q being pulled LOW. Also, since no current path exists between the $\overline{Q}$ output and ground, $\overline{Q}$ is pulled HIGH.

Similarly, if differential input signal C is at logic 0, while A and B are at logic 1, transistor t_{C2} is ON. A current path exists between the collector of t_{A2} and ground, causing the Q output to be pulled LOW ($\overline{Q}$ is again HIGH).

It can be observed that the output Q is HIGH (and its complement $\overline{Q}$ is LOW) if, and only if, all the differential input signals A, B and C are at logic 1, i.e. the gate performs the AND function.

The logic swing of the gates is defined by the load resistors, R_L, and the gate current, I_E, and is nominally 160mV. By selecting between alternative gate current-resistor 'pairs' different speed/power options are available.

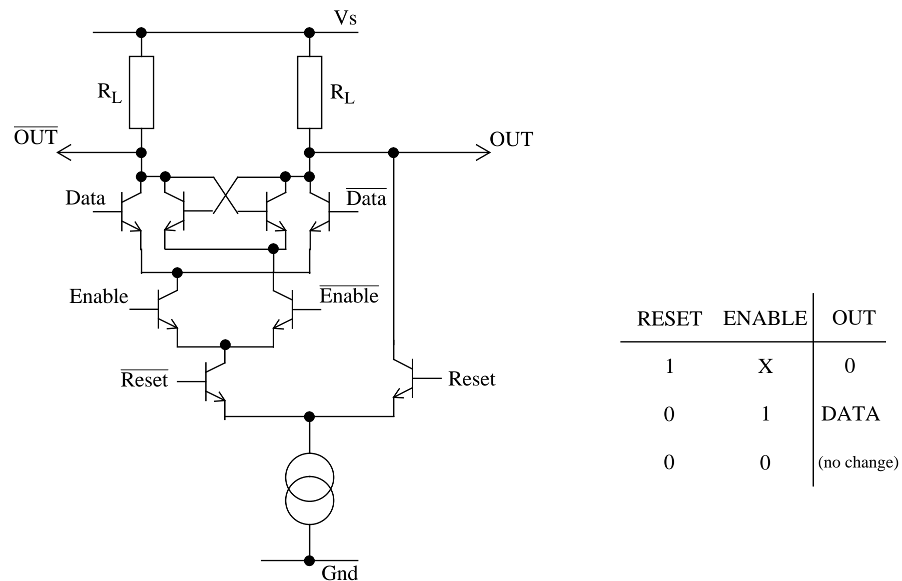
Due to the differential switching tree arrangement, many complex logic functions can be incorporated into a single gate structure. Two example functions, a 4:1 Multiplexer and a Transparent Latch with Reset are shown in Figures 11 and 12.



S1	S0	OUT
0	0	A0
0	1	A1
1	0	A2
1	1	A3

Figure 11 : 4:1 Multiplexer.

The S1 and S0 inputs, at level 1 and level 2 respectively, uniquely select one of the level 3 inputs (A3-A0) for passing to the output.



RESET	ENABLE	OUT
1	X	0
0	1	DATA
0	0	(no change)

Figure 12 : Transparent Latch with Reset.

The Reset signal overrides all other inputs and so is at level 1 - when Reset is asserted the Data/Enable switching tree is not active and the output signal is driven to logic 0 (OUT = LOW, $\overline{OUT}$ = HIGH). When Reset is deasserted, the Data input is passed through to the output (when Enable = 1) or the latch holds the output stable (when Enable = 0). Data storage for the latch is provided by the cross-coupled pair of transistors at level 3.

In summary, the advantages of MDCML are:

- Non-saturating switching transistors and very small voltage swings allow very high speed operation.
- Differential operation removes the requirement for temperature-compensated voltage reference circuits (needed in ECL).
- Increased noise immunity and enhanced resilience to supply voltage fluctuations, temperature variations and IR drops because of the excellent Common-Mode Rejection Ratio.

- The multiple levels of switching transistors results in a high functionality of the standard cells.
- The use of differential signals removes the requirement for inverters (a signal is inverted by simply “swapping the wires”), which may represent up to 20% of all the gates in a system [GPS88].
- The high impedance of the long-tailed pair arrangement enhances high fan-out operation.

However, disadvantages include:

- High static power dissipation - although MDCML can be operated at 3V and the small voltage swings employed result in very small currents when compared with ECL (MDCML - 90μA, ECL ~ 1mA).
- Extra silicon area and power is required for level shifting circuits.
- The routing area needed may be increased by a factor of two, since two wires are needed for each signal. A CAD system may require more sophisticated routing software since, to preserve the common-mode rejection characteristics of differential logic, the two signal wires must be routed as a single entity.

6. Verilog Modelling of MDCML

6.1 Requirement for Accurate Model of System

The utility of a simulation model of a complex digital system is ultimately determined by the extent to which that model closely reflects reality. A model that is simple and easy to manipulate is of little benefit if it does not mirror the actual switching characteristics of the implementation technology.

Circuit level simulation is normally the lowest level of simulation used in the design of an electronic system and is usually performed on circuits consisting of a few tens of components: transistors, resistors, capacitors etc. [Russ85]. The circuit simulation determines the electrical characteristics of the component group which may form a logic gate primitive, for example an AND gate, and may require a few minutes of CPU time.

Circuit simulation of an entire design consisting of many thousands of transistors may be performed in rare circumstances, but generally, the computing resources required make this approach prohibitive. To produce an accurate design simulation, the standard solution is to model the system at a higher level of abstraction based on information gained from circuit simulation of the primitive components [Hill87].

6.2 Determination of Electrical Characteristics of MDCML

The switching characteristics of MDCML logic primitives are determined by circuit simulation of the arrangement of transistors and associated component models required

to produce a particular logic function. The circuit simulation is achieved using HSPICE [Hspi90], a widely-used, commercially-available, development of the original Berkeley SPICE program [Nage73].

For the purposes of demonstrating the simulation procedure, a 2-input AND gate will be used as an example. A circuit diagram of the primitive components used to construct a 2-input AND gate is shown in Figure 13:

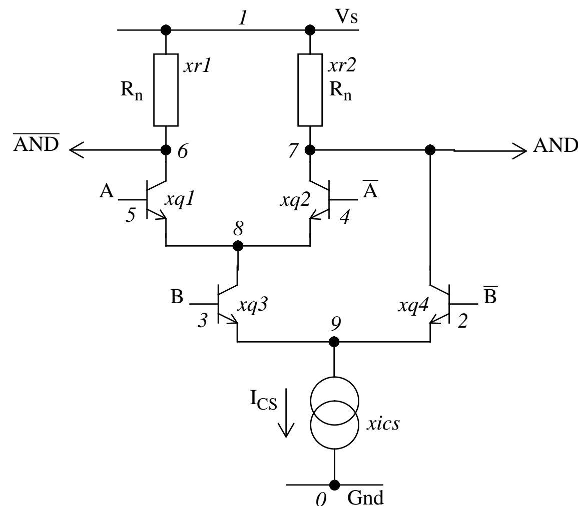


Figure 13 : 2-input AND gate.

A textually-based SPICE model of the circuit is produced:

```
*subckt and2 out Nout i3 Ni3 i2 Ni2 Vs
.subckt  and2  7    6    5  4    3   2   1
xr1   1  6   rn
xr2   1  7   rn
xq1   6  5  8  t20
xq2   7  4  8  t20
xq3   8  3  9  t20
xq4   7  2  9  t20
xics  9  0   cs90
.lib 'Elibbase' rn
.lib 'Elibbase' t20
.lib 'Elibbase' cs90
.ends and2
```

Figure 14 : SPICE model of AND2.

Each component instance is given an instantiation name. In the AND2 gate model of Figure 14, transistors have been labelled, xq1, xq2, xq3 and xq4, resistors have been labelled xr1 and xr2 and the current source is labelled xics. The circuit connections are

specified in terms of numbered nodes and the name of the model primitive used for each component is indicated. For example, “**xq3 8 3 9 t20**” specifies a transistor with the instance name xq3, which has its collector, base and emitter connected to nodes 8, 3 and 9 respectively and has the circuit behaviour defined by the t20 transistor model.

The propagation delays from each of the inputs, A and B, to the output are measured for both rising and falling edges. Since both phases of the signal are available in differential logic, delays are measured from the input crossover point to the output crossover point. The A (level 3) input, B (level 2) input and output waveforms are shown in Figure 15.

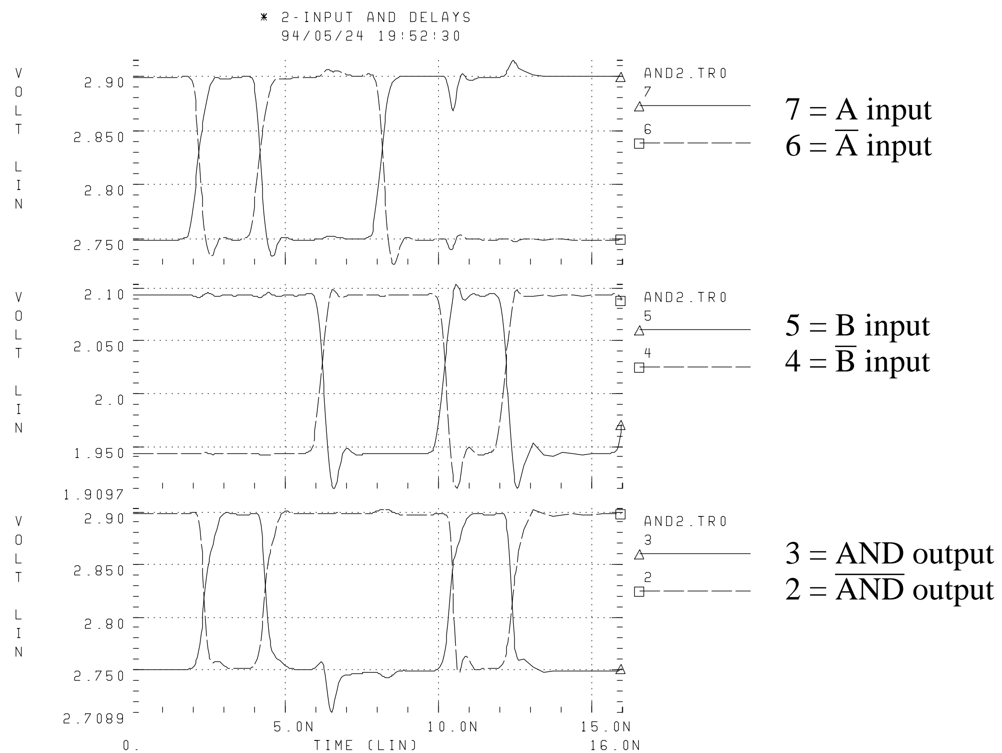


Figure 15 : 2-input AND gate SPICE waveforms.

The measured propagation delays for an unloaded 2-input AND gate are:

A rising -> OUT rising = 178ps

A falling -> OUT falling = 178ps

B rising -> OUT rising = 241ps

B falling -> OUT falling = 193ps

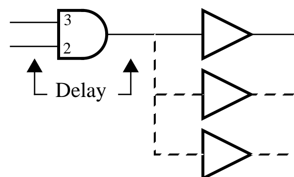
The results shown in Figure 15 indicate that:

i) The level 3 (A input) propagation delay is less than the level2 (B input) propagation delay: the higher levels in the MDCML switching tree switch faster.

ii) The rising and falling delays of level 2 are different. This may be explained by noting that the switching tree is non-symmetrical above the level 2 inputs.

6.2.1 Output Loading Effects

The effects on the propagation delay of loading the gate output are now considered. The output load is provided by the successive addition of **level 3 buffer** circuits. The buffer circuit is chosen for this purpose because no level shifting is required between the AND2 gate output and the input of the buffer. Also, since the buffer circuit has a symmetrical switching tree structure, it should provide an equivalent response to both rising and falling input waveforms. The topology of the test circuit is shown below:



The propagation delay effects of gate output loading were measured for both level 3 and level 2 input signal changes, and for both rising and falling edges. The following results were obtained (all times measured in picoseconds):

Additional O/P Load	LEVEL 3		LEVEL 2	
	Rising	Falling	Rising	Falling
0	178	178	241	193
1	205	205	262	220
2	232	233	283	249
3	262	263	307	281
4	294	296	333	317
5	330	332	362	358
6	370	372	404	404
7	412	412	440	444

A graph of additional delay against additional load was plotted (see Figure 16).

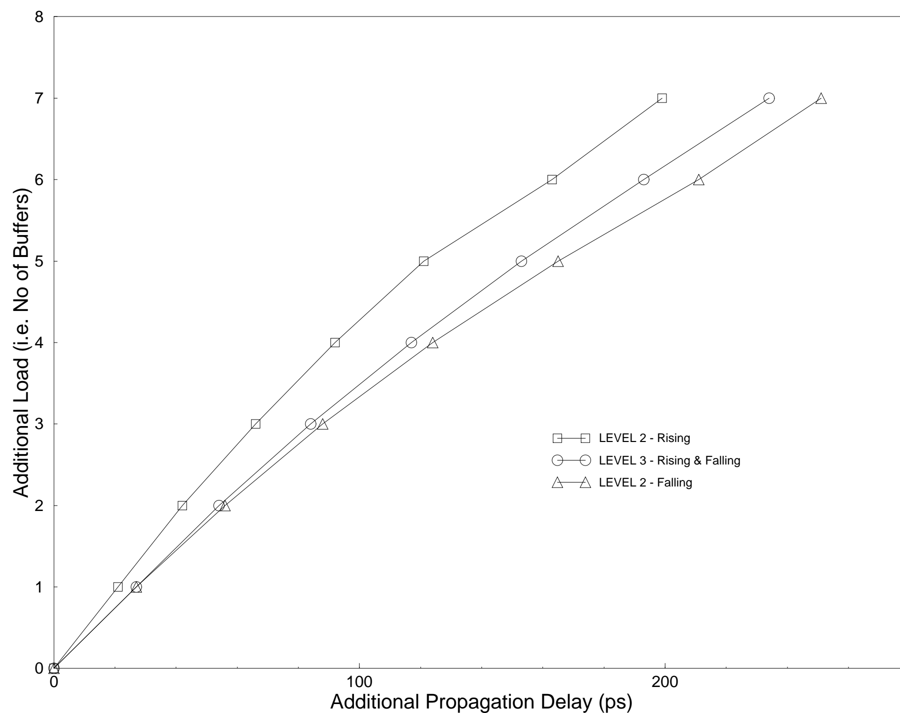
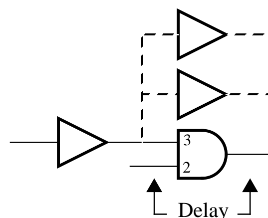


Figure 16 : Graphs of Additional Load vs Additional Delay.

The graphs can be approximated to a straight line through the origin. The conclusion drawn from the results is that each additional load applied to the output of a 2-input AND gate adds around 30ps to the propagation delay for both input levels and for both rising and falling edges.

6.2.2 Input Drive Effects

When the *preceding* gate is heavily loaded, this can have an effect on the propagation delay of the gate in question. This is as a result of the input signal drive capability being 'shared' between several *successor* gate inputs. A **level 3 buffer** circuit was again used as the (preceding) gate load in the following configuration:



The effects on propagation delay were measured for each level and for rising and falling input signal changes. The following results were obtained:

Drive Sharing	LEVEL 3		LEVEL 2	
	Rising	Falling	Rising	Falling
0	178	178	241	193
1	188	189	244	196
2	199	200	248	199
3	210	211	251	202
4	222	222	254	205
5	232	233	258	207
6	242	241	262	210
7	250	248	265	213

A graph of additional delay against extent of drive sharing was plotted (see Figure 17).

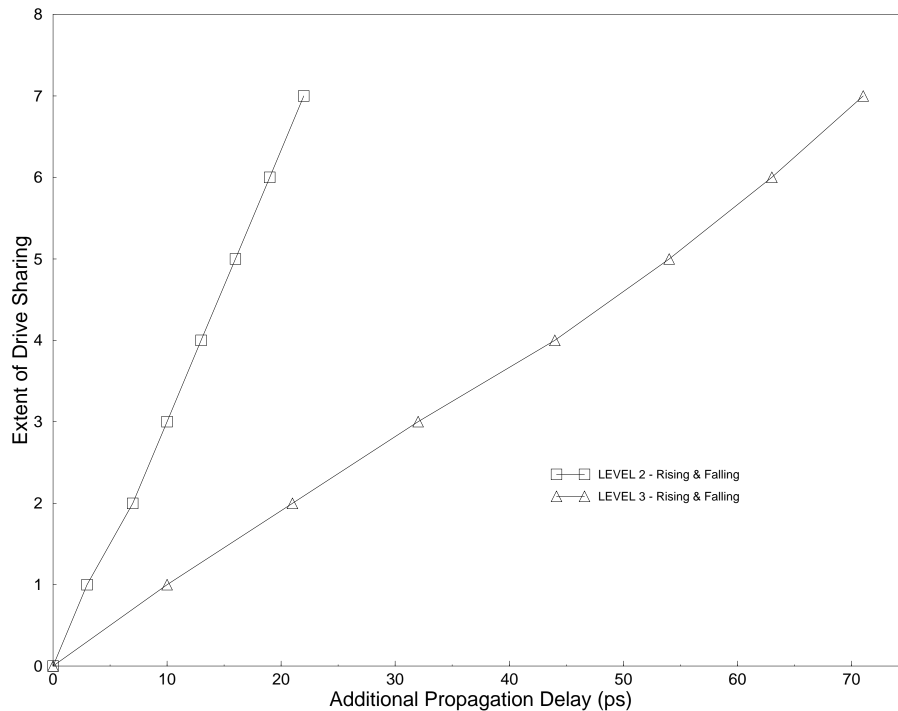


Figure 17 : Graphs of Additional Delay vs Extent of Drive Sharing.

The conclusion is that the effect of input drive sharing on the 2-input AND gate is different for each of the input signal levels: for level 3, 10ns is added to the propagation delay for each gate sharing the drive, for level 2 the delay is only increased by 3ns. This suggests that level 2 signals have a greater drive capability than level 3 signals.

Also, the effect on gate propagation delay of input drive sharing is less significant than the effect of output loading.

6.3 Production of Verilog Model

The information obtained from the circuit simulation may then be used to generate a model capable of being simulated at a higher level of abstraction. In this case, a behavioural or gate-level model is produced using the Verilog Hardware Description Language.

On initial examination of the simulation data, two points emerge regarding the switching characteristics of the MDCML 2-input AND gate. Firstly, there are different propagation delay for the rising and falling signal changes of the level 2 input and, secondly, the delays differ for the different input levels.

For gate-level modelling in Verilog, both rising and falling propagation delays may be specified for each logic primitive. For behavioural modelling, both input signal edges may be detected using the "**always** @ (*posedge* ...)/**always** @ (*negedge ...)"* construct. In this example, however, only a single propagation delay will be specified for each input level (for either signal transition direction) to maintain the model simplicity.

The worst-case delay will be used for each level:

A (level 3) -> out = 178ps

B (level 2) -> out = 241ps

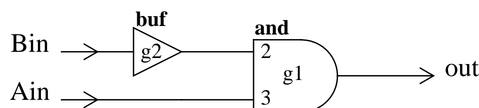
Both a behavioural and a gate-level model of the 2-input AND gate are produced to demonstrate how a logic primitive may be modelled. Also, two approaches to modelling the different input level propagation delays can be shown.

Considering the behavioural module first:

```
module and2 (out, Ain,Bin);
`timescale 1ps/1ps

`define and2A_delay 178
`define and2B_delay 241

input Ain, Bin;
output out;
reg out;

always @ (Ain)
    #(`and2A_delay) out = Ain & Bin;

always @ (Bin)
    #(`and2B_delay) out = Ain & Bin;

endmodule
```

In the behavioural example, the module is triggered when there is a change in any of the input signal values. Depending on whether the input change is at level 3 (Ain) or level 2 (Bin), the output is specified to change (if an output value change is warranted) after a different time interval, *and2A_delay* or *and2B_delay*. In this manner, the different propagation delays from input to output of the different levels are modelled.

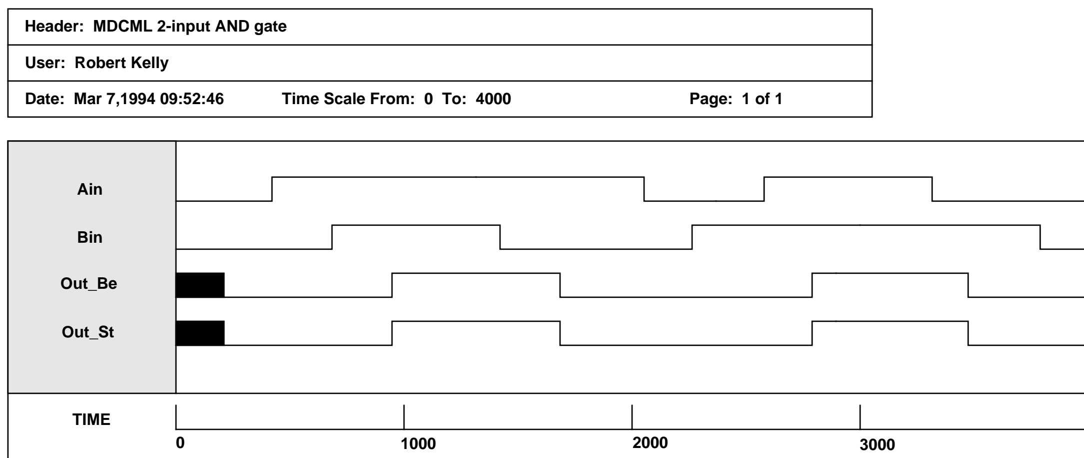
The gate-level, or structural, module is based on the **and** and **buf** logic primitives used in the following configuration:




```
module and2 (out, Ain,Bin);
`timescale 1ps/1ps

`define and2A_delay 178
`define and2B_delay 241

input Ain, Bin;
output out;
wire delb;

and #(`and2A_delay)               g1 (out, Ain,delb);
buf #(`and2B_delay - `and2A_delay) g2 (delb, Bin);

endmodule
```

A single propagation delay is specified for the **and** logic primitive; the minimum of the A and B propagation delays, which is *and2A_delay*. An additional delay is encountered by a B input signal change and this is modelled by providing a **buf** (buffer) logic primitive. The **buf** element has a propagation delay equivalent to the difference between the propagation delays of the two input levels (A and B). In this way, an A input change will propagate to the output after the *and2A_delay* time through the **and** primitive. Also, a B input signal change will propagate to the output after the '*and2B_delay - and2A_delay*' time through the **buf** primitive plus the *and2A_delay* time through the **and** primitive, i.e. a total propagation delay time of *and2B_delay*.

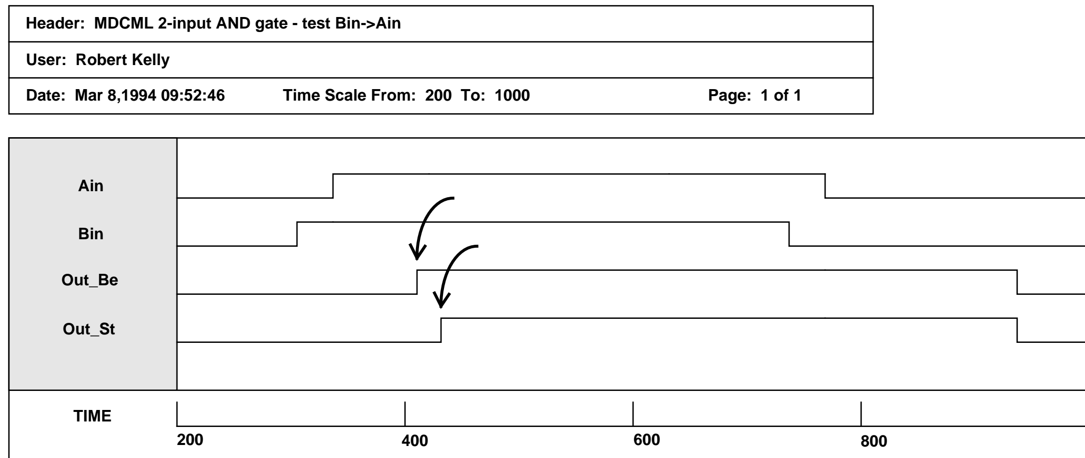
An example of the waveforms produced by both the behavioural (AND2_Be) and structural (AND2_St) modules when simulated in Verilog using the same input stimuli is shown below:



6.3.1 Accuracy Comparison

The MDCML 2-input AND gate exhibits differing propagation delays to input changes occurring at the different levels. This suggests that the operation of the Verilog models of the AND2 gate may be sensitive to simultaneous or nearly simultaneous changes in the input signals. The two models of the AND2 gate were simulated under these conditions and a problem was discovered which can be observed in the following

waveforms:



The output waveforms of the two AND2 modules indicates that the behavioural and gate-level models react differently to the specified input stimulus. In particular, the response to a B input change closely followed by an A input change must be examined for each of the models.

For the behavioural module, when the A input change occurs **(always** @ (Ain)), the output assignment expression, out = Ain & Bin, will be evaluated. At this point, both the Ain and Bin inputs are HIGH and so the output value will be scheduled to change after the propagation delay time of the Ain input, #(‘and2A_delay) out = Ain & Bin.

For the gate-level (structural) module, when the Ain input change occurs, the previous change of the Bin input has not yet propagated through the **buf** primitive. So, at this point in time, the A input to the **and** primitive is HIGH, but the B input to the **and** primitive is LOW. Therefore, the Ain input change does not directly affect the output of the **and** primitive in this situation. Some time later, when the effect of the Bin input change has propagated through the **buf** primitive, both inputs to the **and** primitive will be HIGH and the output will be scheduled to change.

In summary, for the behavioural model, the Ain propagation delay takes priority, whereas for the gate-level model, the Bin propagation delay takes priority. The question is then: which model most closely reflects reality? To provide the definitive answer, a

HSPICE circuit simulation is performed with the appropriate input stimulus. The results are shown below in Figure 18.

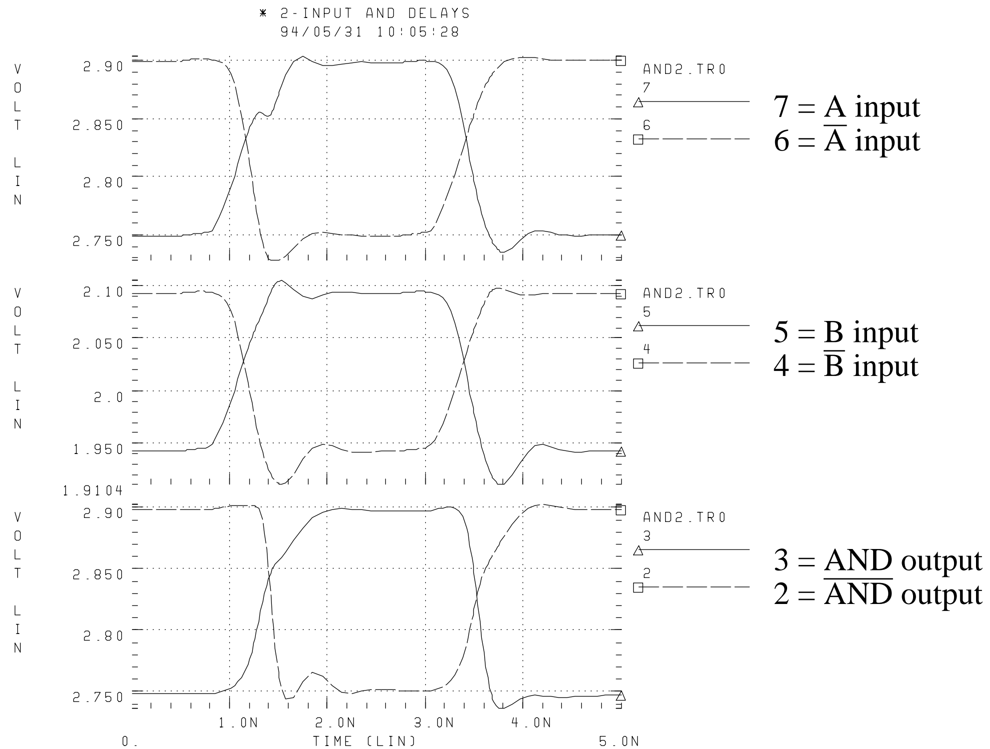


Figure 18 : Ain after Bin SPICE waveforms.

The Ain input change occurs 21.6ps after Bin for the rising transition and 22.3ps after Bin for the falling transition.

The measured propagation delays are:

A rising	->	OUT rising	=	238ps
B rising	->	OUT rising	=	260ps
A falling	->	OUT falling	=	113ps
B falling	->	OUT falling	=	135ps

Considering the rising transitions of the A and B inputs, the actual measured output propagation delay is most closely modelled by the gate-level module. That is, the B input signal change tends to take priority. The output propagation delay of the falling input transitions seems to be anomalous; since the measured delay is much less than either

the normal A or B input falling transition delays. This behaviour may be as a result of both input changes simultaneously tending to force the output LOW.

Since the gate-level, structural, module gives the most accurate modelling behaviour of the 2-input AND gate, this module is chosen as the basis for the AND2 logic gate in the MDCML Verilog component library. The full component library is constructed by considering each logic gate function in a similar manner.

6.3.2 Continuous Assignment

The CPU resources required for the simulation of a large scale digital system can be significant, even when the design is simulated at higher levels of abstraction. The Verilog HDL provides a mechanism for accelerating the performance of the modelling constructs by applying a technique known as *continuous assignment*. Continuous assignment may be used to increase the simulation performance of models by directly assigning values to outputs of primitives based on the values currently on the inputs.

The gate-level model of the 2-input AND gate can be replaced by a continuous assignment version which increases the simulator performance:

```
module and2 (out, Ain,Bin);
`timescale 1ps/1ps

`define and2A_delay 178
`define and2B_delay 241

input Ain, Bin;
output out;
wire delb;

assign #(`and2A_delay)             out = (Ain & delb);
assign #(`and2B_delay-`and2A_delay) delb = Bin;

endmodule
```

Here the **assign** statement replaces the instantiations of the **and** and **buf** primitives in the gate-level model.

The technique is restricted to logic primitives that are:

- ❑ purely combinatorial, i.e. the primitive does not contain any internal state.
- ❑ based on very simple logical operations.

6.3.3 Net Delays

The issue of how the output loading and drive sharing information should be incorporated into the Verilog model of the system is now examined.

It can be noted that, essentially, the causes of the propagation delays of an interconnected system of component modules can be divided into two types. The first type, *Elemental* delays, concern the direct operation of the basic logic function, i.e. the propagation delay through the unloaded gate. The second type, which can be called *Topological* delays, cause an additional propagation delay to be applied to each basic component depending on the nature of its interconnections, both input and outputs, with the rest of the system, i.e. the fan-in and fan-out of the individual gates.

Verilog provides a facility whereby delay values may be assigned to individual nets (wires) connecting system modules (known as **net** delays). This would appear to be an ideal method of managing the additional propagation delay effects of the system interconnections. The basic component modules would incorporate the Elemental delays and the interconnecting nets would include the Topological delays.

For completeness, the system models should incorporate the effects of input drive-sharing and output loading. However, for a reasonably large design, calculating the additional gate propagation delay due to these effects by hand would be tedious - some form of automatic netlist generation is required (this is best achieved in conjunction with a schematic design-capture system).

In addition, the circuit delay values due to track capacitance (measured during the physical layout design process) could easily be backannotated into the Verilog simulation model by modifying the **net** delay values. However, until the actual physical layout of

the design is accomplished, and real track capacitance values can be used to generate real interconnection delay values, the MDCML Asynchronous ARM Verilog model will use only elemental delays.

7. MDCML Asynchronous ARM

7.1 ARM Architecture

7.1.1 Overview

The Advanced RISC Machine (ARM) is a general purpose 32-bit microprocessor architecture based on the Reduced Instruction Set Computer (RISC) principle of a simple, regular instruction set allowing fast and efficient decoding [Furb89,VLSI90]. Together with a three stage (fetch, decode, execute) execution pipeline, this results in a high instruction throughput. The ARM uses a load/store architecture with a register-oriented instruction set.

The ARM6 (the target architecture of the MDCML Asynchronous ARM) has a 32-bit data space and a 32-bit address space [ARM91](see Figure 19). All instructions are one word (32-bits) and all data processing operations are performed on word quantities. Byte quantities (in addition to words) can only be specified for load and store operations.

The ARM6 may be executing instructions in one of six processor modes:

- User - normal program execution.
- Supervisor - protected mode for operating system support.
- IRQ- normal interrupt handling.
- FIQ - ‘fast’ interrupt handling (for external data I/O).
- Abort - data or instruction prefetch abort.
- Undef - undefined instruction execution.

Most applications programs execute in User mode, the other (privileged) modes are entered to service interrupts or handle processor exceptions.

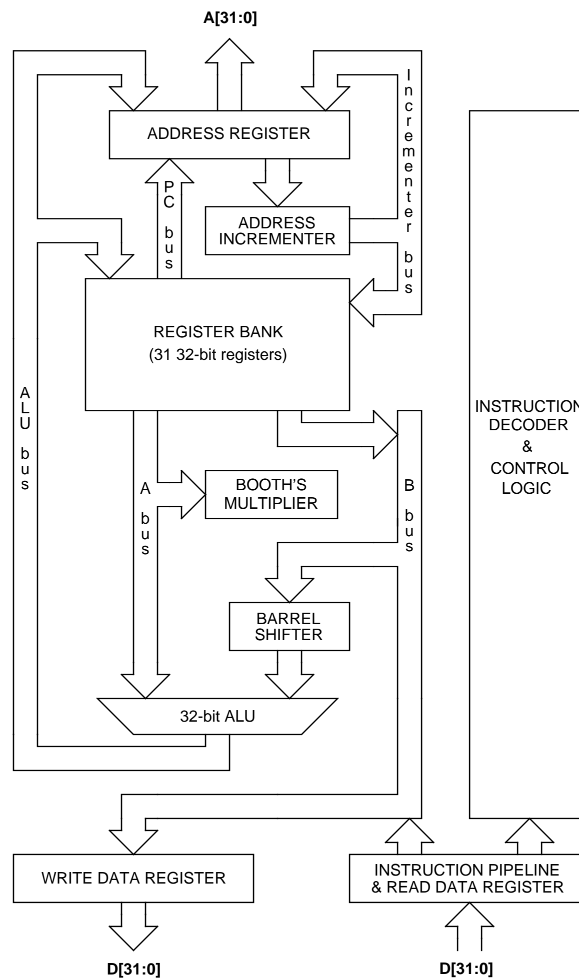


Figure 19 : ARM6 Block Diagram.

A total of 37 registers are provided, 31 general purpose registers (each 32-bits wide) and 6 status registers. The registers partially overlap such that, depending on the current processor mode, only 15 general purpose registers (R0 - R14) and R15 holding the Program Counter (PC) are ‘visible’. In all modes the Current Program Status Register (CPSR), which contains the condition code flags and the current mode bits, is visible and in the privileged modes the Saved Program Status Register (SPSR) is also visible. R14 is used as the subroutine link register (receiving a copy of the PC return address on executing a Branch and Link instruction).

The fast interrupt mode (FIQ) has seven 'banked' registers (R8 - R14) and all privileged modes have banked R13 (stack pointer) and R14 (subroutine link) registers. There is a SPSR (loaded with the CPSR on exception entry) for each of the privileged modes.

7.1.2 Instruction Set

The ARM instruction set consists of ten basic instruction types. All ARM instructions are conditionally executed based on the value of the N, Z, C and V flags in the CPSR. The conditions *always* (AL) and *never* (NV) also exist. Conditional execution of all ARM instructions seeks to improve processor performance by removing the need for small-offset forward branches which therefore maintains the execution pipelining.

The data processing instructions can be divided into two groups: those concerned with logical operations (**AND, EOR, ORR, BIC, MOV, MVN, TST, TEQ**) and those performing arithmetic operations (**ADD, ADC, SUB, RSB, CMP, CMN**). This class of instruction also contains an **S** bit which indicates whether the condition codes should be set based on the result of the specified operation. Since the ARM architecture contains a barrel shifter connected to one of the input operand buses of the Arithmetic Logic Unit (ALU), it is possible to perform various shift functions on one of the input operands before the specified data operation is applied. A subset of the data processing type, the **MRS**/**MSR** instructions provide access to the CPSR and the SPSR: the **MRS** instruction moves the contents of the CPSR or SPSR into a register and the **MSR** instruction moves a register value into the CPSR or SPSR.

The branch (**B**) and branch-with-link (**BL**) instructions allow the PC to be modified by adding a signed offset. A 'jump' instruction can also be achieved by using the **MOV** instruction to load the PC (R15) directly with an immediate or register value.

A multiply (**MUL**) or multiply-accumulate (**MLA**) instruction uses a 2-bit Booth's algorithm to perform integer multiplication, the multiply-accumulate form adds a third operand register value to the result of the basic two input register multiplication.

Two instruction types are concerned with moving data between registers and memory. The **LDR/STR** data transfer instructions move a single byte or word of data. The **LDM/STM** block data transfer instructions are used to move any subset of the currently visible register set.

The software interrupt (**SWI**) instruction is used to enter supervisor mode in a controlled manner and the single data swap (**SWP**) instruction is used to swap a byte or word quantity between a register and memory as an 'atomic' (uninterruptable) operation - this facility provides the basis for multiprocessing semaphore support.

Three further instruction types are used in the context of coprocessor interaction and will not be discussed further.

7.2 MDCML Asynchronous ARM

7.2.1 Overview

The high-level design of the MDCML asynchronous ARM will closely follow that of the AMULET1 [Furb94b]; an asynchronous ARM microprocessor developed for CMOS technology within the ESPRIT OMI-MAP project involving the AMULET group at Manchester University. The TAM-ARM project will also consider if any of the design enhancements proposed for the CMOS successor to AMULET1, in the light of experience gained while producing the original prototype, will be appropriate for the MDCML implementation.

Much of the architectural design information presented in this chapter is derived from [Pave94], a Ph.D. thesis of one of the principal design team members.

The internal structure of the MDCML asynchronous ARM is shown in Figure 20 overleaf.

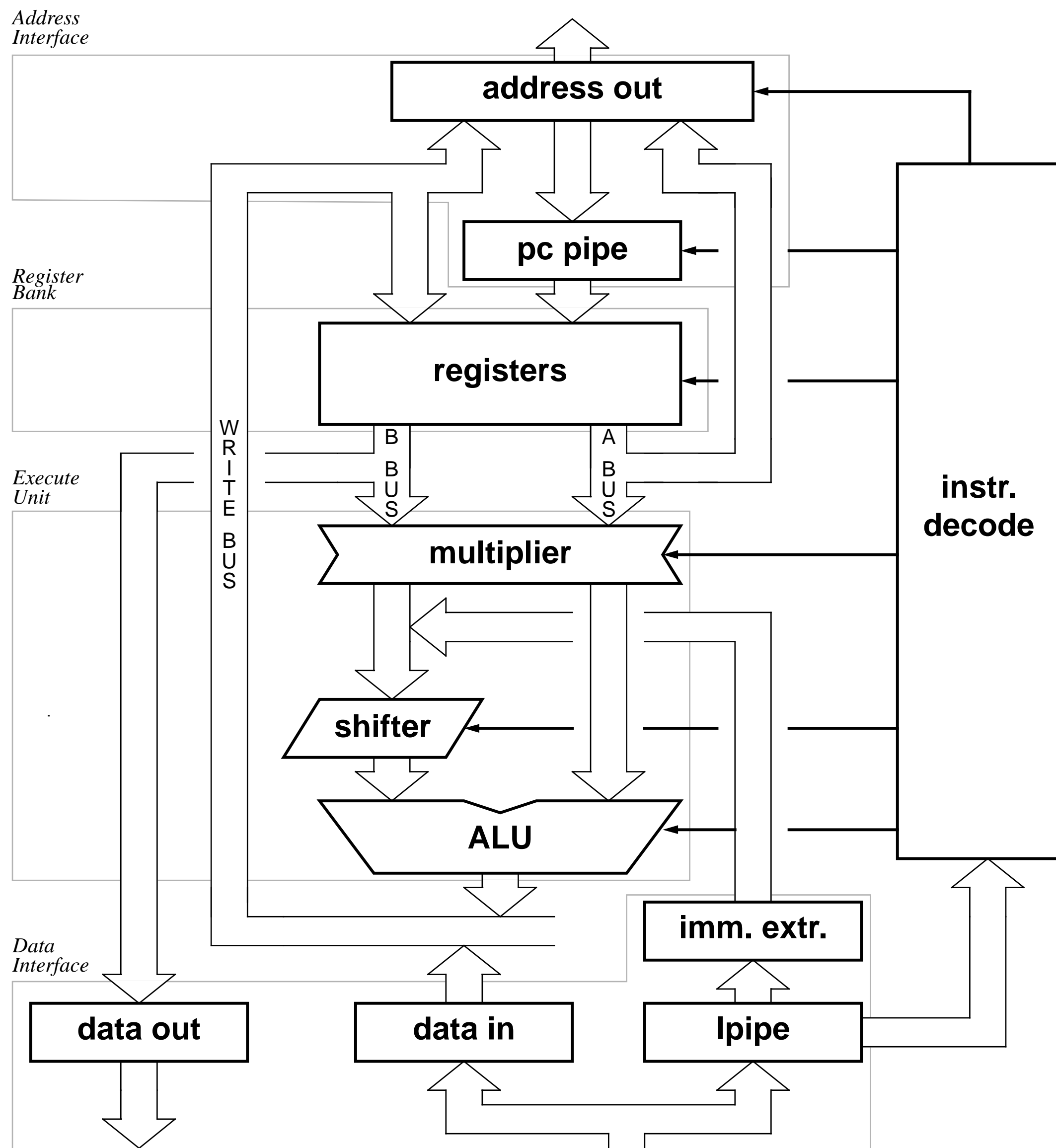
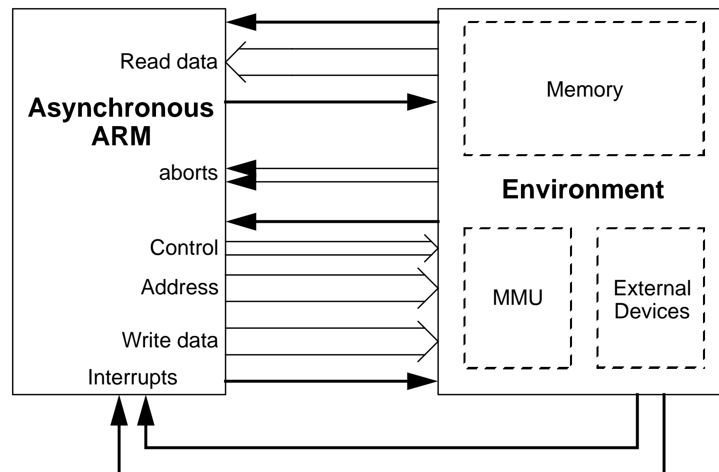


Figure 20 : Internal Processor Organisation.

Since the MDCML asynchronous ARM will be a preliminary demonstrator of an asynchronous bipolar implementation of the ARM6 architecture, several features have not been incorporated into the MDCML asynchronous ARM architecture due to system design time constraints. Unimplemented features include the class of instructions used to manage coprocessor interaction, support for 26-bit mode operation (an instruction-set backwards-compatibility issue) and the MLA (Multiply-with-Accumulate) instruction.



The MDCML ARM will employ a transition-signalling bundled-data interface between the asynchronous processor and the external memory subsystem involving the following signals:

- An output bundle containing the requested memory address, associated control signals and, possibly, a write data value (if a write operation is specified).
- An input bundle containing a data value (which may be an instruction) read from memory and the bundled-data protocol control signals.
- A memory abort response. Every data access to memory requires a response signal to indicate whether the access will successfully complete. This allows the processor to support a *virtual memory* system.

A processor reset and level-sensitive interrupt request signals complete the MDCML asynchronous ARM connections to the external environment.

The structure of the Execution Pipeline, which includes the Register Bank, Execution Unit and Instruction Decode, is outlined in Figure 21 overleaf. In particular, the typical micropipeline structure of Event Registers (shown as shaded boxes) interposed by computational logic can clearly be seen. The pipeline operation is controlled by the transition signalling protocol operating between the event registers and functional blocks, but the details have been omitted from the diagram in the interests of clarity.

The Primary Decode provides the entire decoding for the Register Bank control signals and a partial decode for the Execution Unit function blocks. Note that the decode and control signals are also pipelined, but this does not imply that the datapath and control operate in lockstep. Control signals and data values only synchronise at the appropriate functional unit.

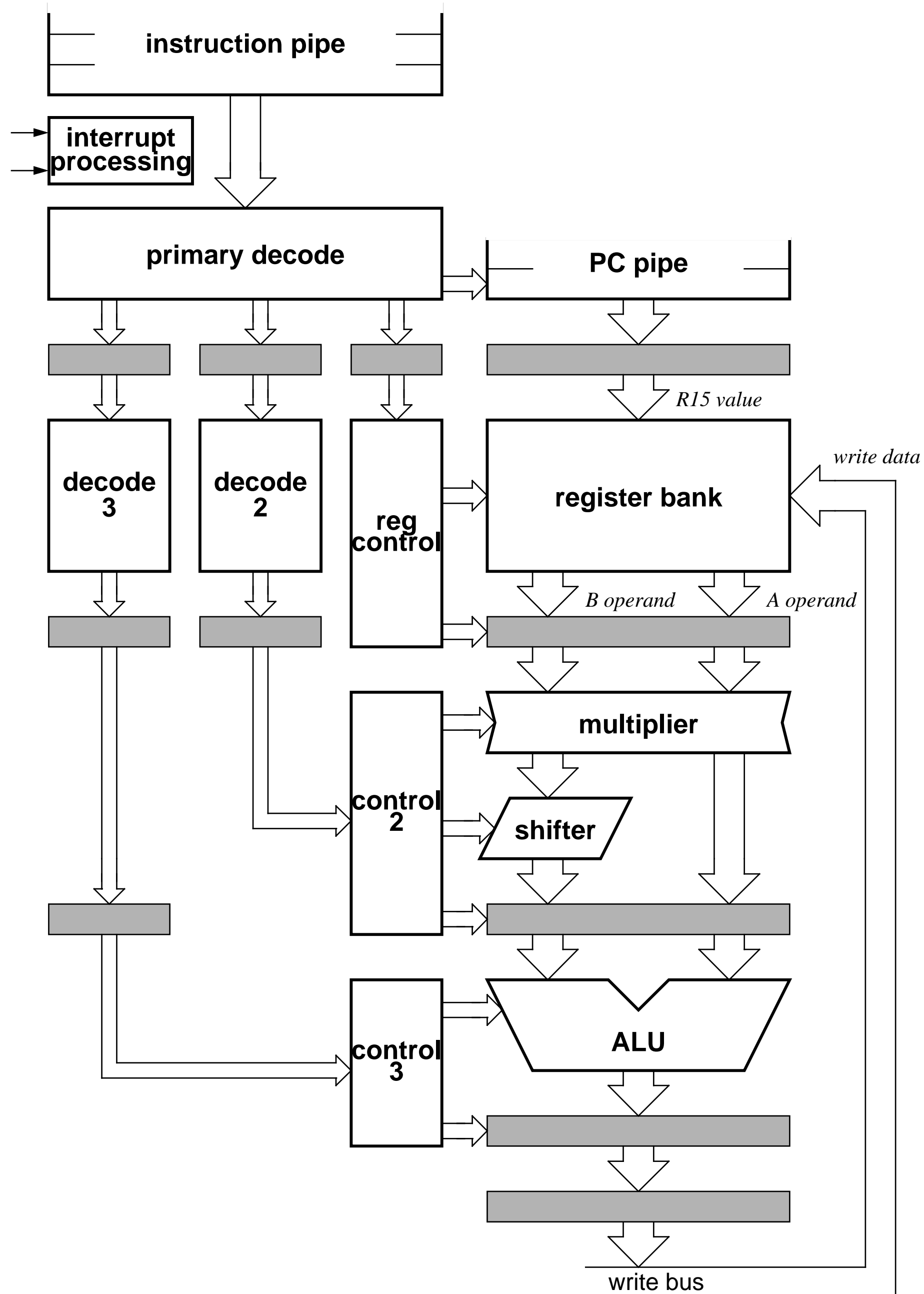


Figure 21 : Micropipelined Structure of the Execution Pipeline.

7.2.2 Register Bank

The register bank provides the storage required for the general-purpose registers and program status registers defined by the ARM architecture. For the execution of a typical instruction, one or two operands are read from the register bank onto the A and B buses. They are then subjected to some logical or arithmetic operation to yield a result, which is normally written back to the register bank via the W bus. To improve overall CPU performance, pipeline operation is employed whereby several instructions may be in different phases of execution. At a certain instant in time, the operands of one instruction may be undergoing an ALU operation, while simultaneously, the operands of the next instruction are being read from the register bank and the ALU result of the previous instruction is being written back to the register bank.

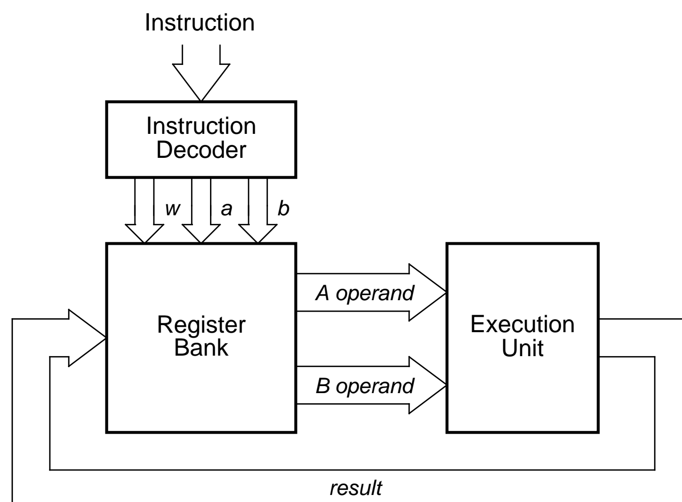


Figure 22 : Register Bank Operation.

In an asynchronous design context, pipelined instruction execution with concurrent read and write access to registers presents a number of problems with regard to coherent register operation:

- ❑ Due to execution phase pipelining, multiple register write operations may be outstanding. The register bank control logic must maintain a record of the correct sequence of write register addresses.

- An operand read action may be requested from a register that has a write operation pending. The register read needs to be suspended until the write actually occurs (the register read should get the new register value).
- Asynchronous read and write operations on the same register may interact unpredictably.

These problems may be solved by storing the write (destination) register addresses in a FIFO (First-In, First-Out queue). After the register bank is accessed during the initial stages of instruction execution, to provide the operands, the destination register address is entered into the FIFO. When the instruction result value eventually arrives back at the register bank, the destination (write) address is at the end of the FIFO. The depth of the FIFO determines the number of register write operations that can be outstanding.

On every read access to the register bank to obtain the instruction operands, the Write Address FIFO is examined. If either of the read addresses is found to match any of the write addresses in the FIFO, the read operation for that particular register stalls until after the write operation on the register has completed. Once the operands have been successfully read from the register bank, the destination (write) register address for the instruction result is entered into the Write Address FIFO. The FIFO effectively provides a 'locking' function on the register bank to prevent Read-after-Write register hazards - hence the alternative name for the Write Address FIFO is the Lock FIFO [Pave92]. The asynchronous register bank design is shown overleaf in Figure 23.

The operation of the asynchronous register bank is now described - the associated Verilog waveforms in Figures 24, 25 and 26 show a Read cycle, a Read cycle stalled on a register lock and a Write cycle respectively:

The register **read** request (*R_Req*) arrives (Figure 24) and presents two instruction operand register addresses (*A_addr* & *B_addr*) and a destination (write result) register address (*W_addr*). Additional addressing information indicates the current processor mode and hence the 'visibility' of a particular register set. The *R_Req* signal is stalled

(by the Muller-C gate) until the register bank is able to commence another operand read cycle (indicated by the *Rgo* signal). The A and B bus decoders are then enabled (by *Rdec*) and at the same time the destination address is latched into the W latch.

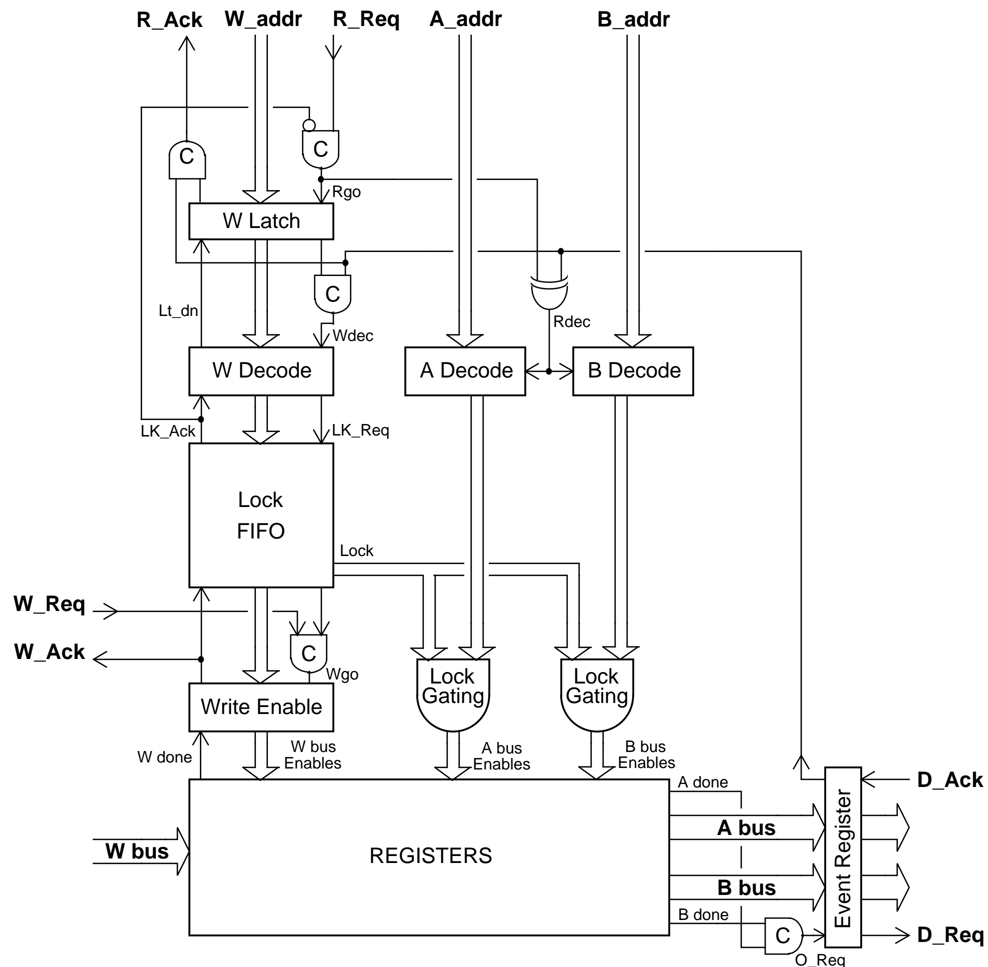


Figure 23 : Asynchronous Register Bank Design.

The decoded read enable signals are gated with the Write Address FIFO lock information for the associated register (*Lock*) - a read will be suspended if the register is <u>locked</u> i.e. a write operation is pending on the register. A read will proceed (*A_enb* & *B_enb* activated) if the register is unlocked, while a read on a locked register must wait until a subsequent write operation clears the lock.

Once both read operations have completed (both *A_done* & *B_done* events have occurred), the operands (*A_bus* & *B_bus*) are latched (by the *O_Req* signal) and the read decoders are disabled. Loading the read output Event Register causes the *D_Req* event

to be signalled to the Execute Unit, indicating that the instruction operands are now available. The write decoder is then enabled (*Wdec*) and the decoded destination register address is entered in the Write Address Lock FIFO (*LK_Req*), thereby locking the register. The write address is stored in the Lock FIFO in decoded form to enable a locked register address to be detected easily (the detail is described elsewhere [Pave91]). Once the Lock FIFO has accepted the new result destination address (*LK_Ack*), the write decoder is disabled and the write address latch is freed (*Lt_dn*). *R_Ack* is signalled and the next instruction can now commence its operand read phase.

The second waveform (Figure 25) shows the effect of a locked register on the read cycle. When the read decoders are enabled (*Rdec*), the B bus register enable (*B_enb*) is activated and the B bus operand read completes (*B_done*, *B_bus* is valid). However, the A bus operand register is locked and therefore the A bus register enables are not activated (*A_enb*). Eventually, a subsequent write operation will clear the register lock (*Lock*), the A bus enables will be activated and the A bus read will complete (*A_done*). The sequence of events following this point is as outlined previously.

The **write** request signal (*W_Req*), Figure 26, indicates that a result value has arrived on the W bus (*W_bus*) for writing into its destination register. When the decoded destination register address (*W_reg*) is available at the output of the Lock FIFO, the write operation can begin (*Wgo*). A control signal (*valid*) also arrives with the data value to provide a facility to clear destination register locks (remove register addresses from the Lock FIFO) without actually writing data into the register bank. This mechanism allows instructions that have failed condition code tests at the ALU to remove write locks from previously ‘reserved’ destination registers.

If the full register write operation is to proceed (*W_ok*), the write bus enables are signalled (*Wr_reg*) and the appropriate register write enable line is activated (*W_enb*) and the data value is written into the register. Once the register write operation has completed (*W_done*), the write bus enables are turned off and the write address is removed from the Lock FIFO (*Lock*) - unlocking the register for subsequent read operations. The

remaining waveforms at the bottom of Figure 26 show a stalled read operation resume once the register lock is cleared.

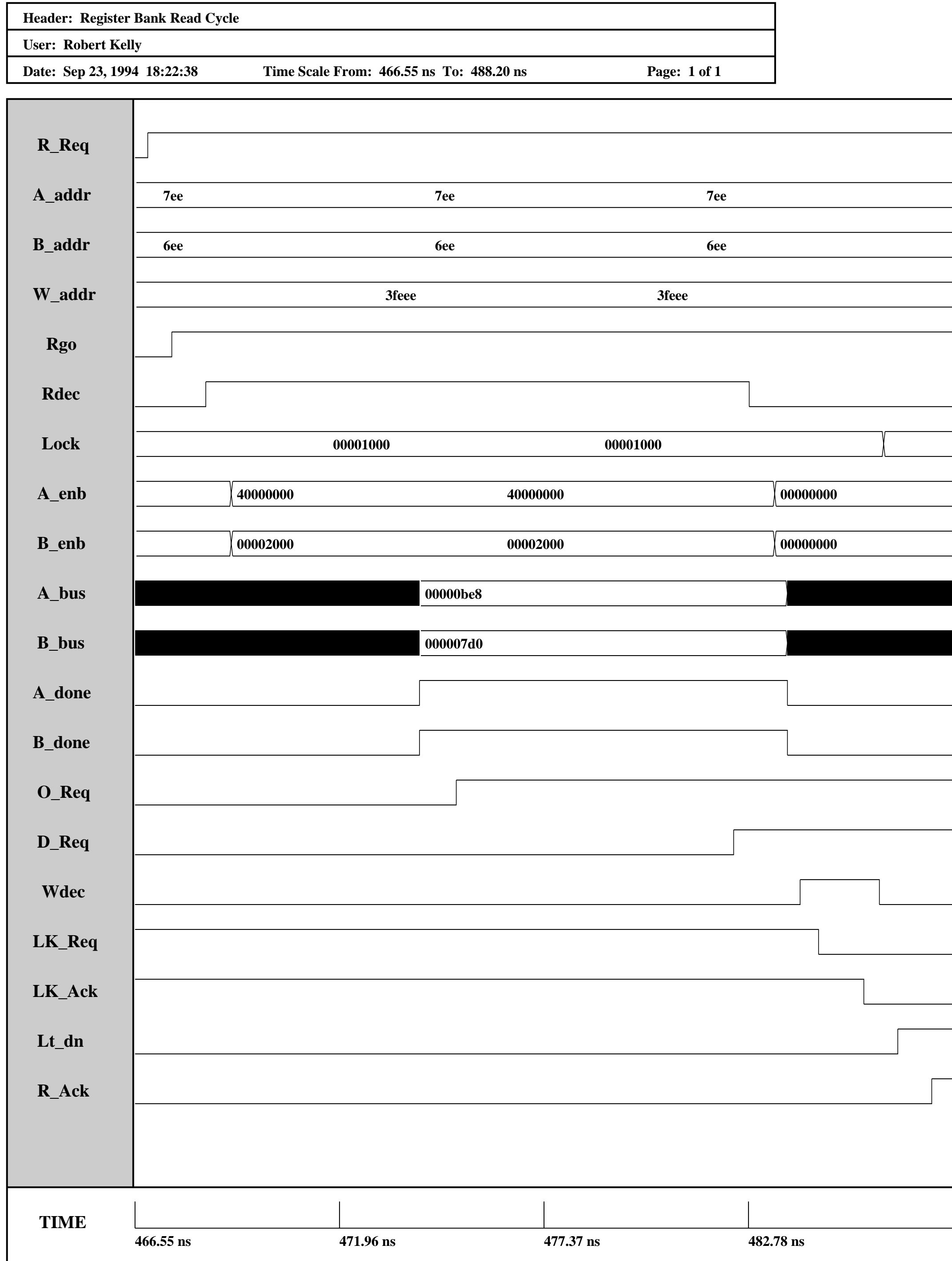


Figure 24 : Register Bank Read Cycle Waveform.

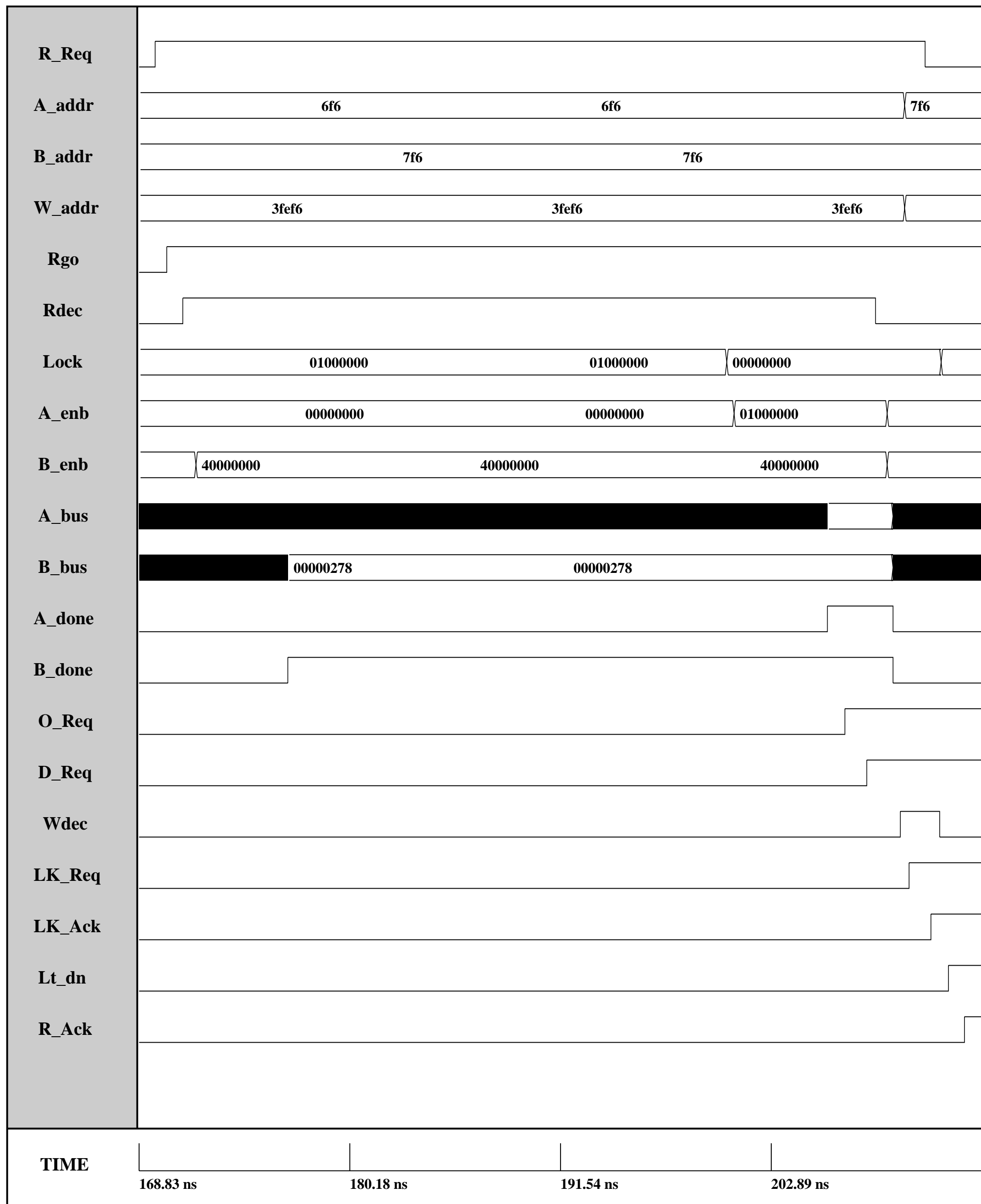


Figure 25 : Register Bank Stalled Read Waveform.

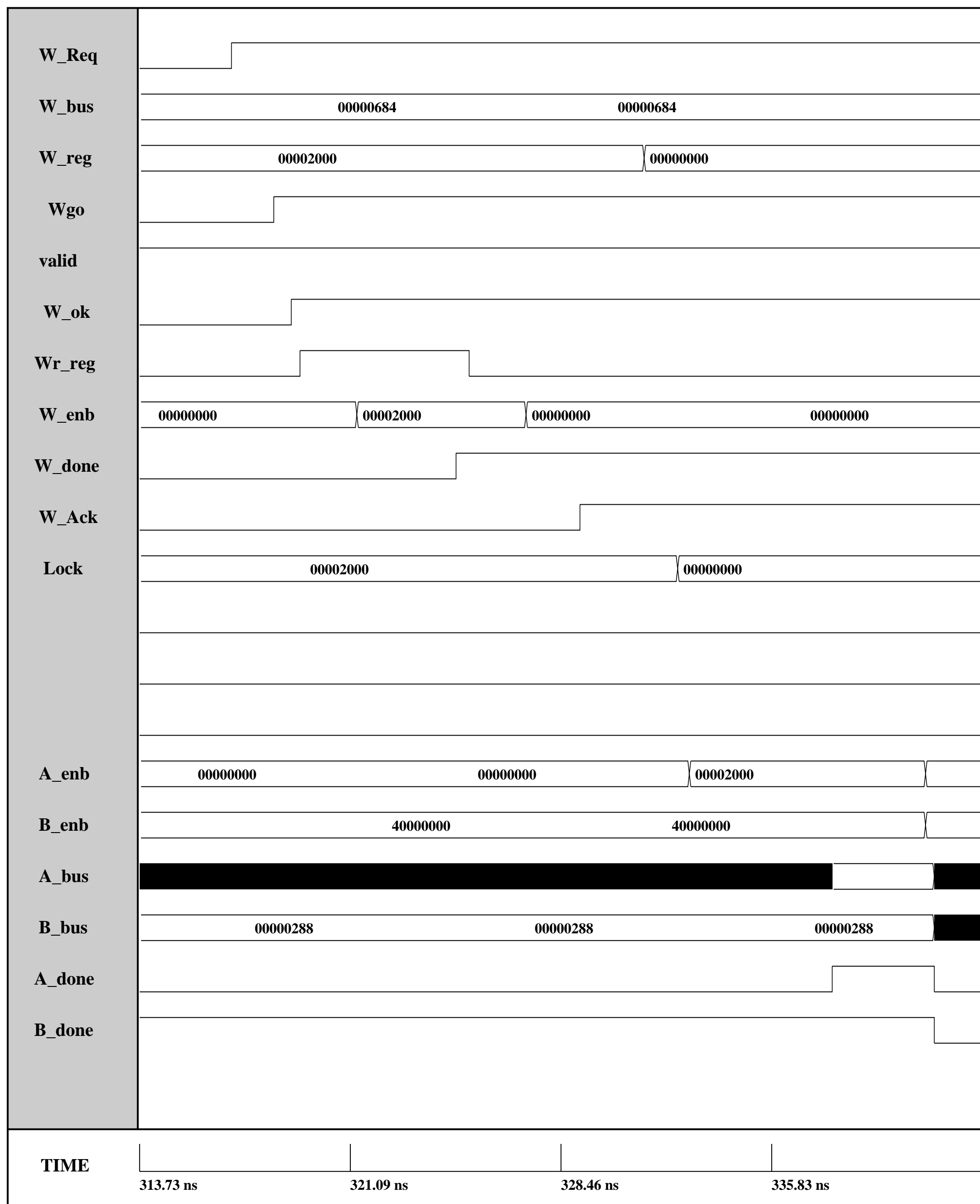


Figure 26 : Register Bank Write Cycle Waveform.

7.2.3 Memory Interface

The interface between the external memory system and the processor is divided into two parts: the **Address Interface** issues all address information to memory and the **Data Interface** is responsible for all data values written to or read from memory.

For a **write** operation, the address generated by the address interface synchronises with the write data value supplied by the data interface before being passed to the memory subsystem.

For a **read** operation, the address generated by the address interface is sent directly to the memory subsystem and control information for the access is passed to the data interface. The control information is examined when the memory read value is supplied to the data interface. The read data value is 'routed' to the correct processor function block destination based on the associated control information.

7.2.4 Address Interface

One of the primary functions of the address interface is to generate sequential addresses for instruction prefetching. The Program Counter (PC) value circulates around a loop containing the Memory Address Register (MAR), an (address) Incrementer and two PC Holding Latches (see Figure 27). Two holding latches are required because of a potential deadlock situation if only one latch was provided. The deadlock occurs when a data transfer request immediately follows the arrival of a new PC value - this is described in detail elsewhere [Pave94, pp126-127]. In each cycle of the PC loop, the PC value is copied into the Memory Address Register where it initiates an external memory instruction read request. After the processor reset signal is deactivated, the Memory Address Register is forced to all zeros and a memory request event is generated causing instruction prefetching (and therefore instruction execution) to begin at memory address (hex) 00000000.

When the address interface is required to generate a memory address for a data transfer operation (either read or write), the PC prefetching loop must be temporarily suspended. Since the prefetch operation is asynchronous with respect to the rest of the processor operation, arbitration is required to gain exclusive control of the address interface resources.

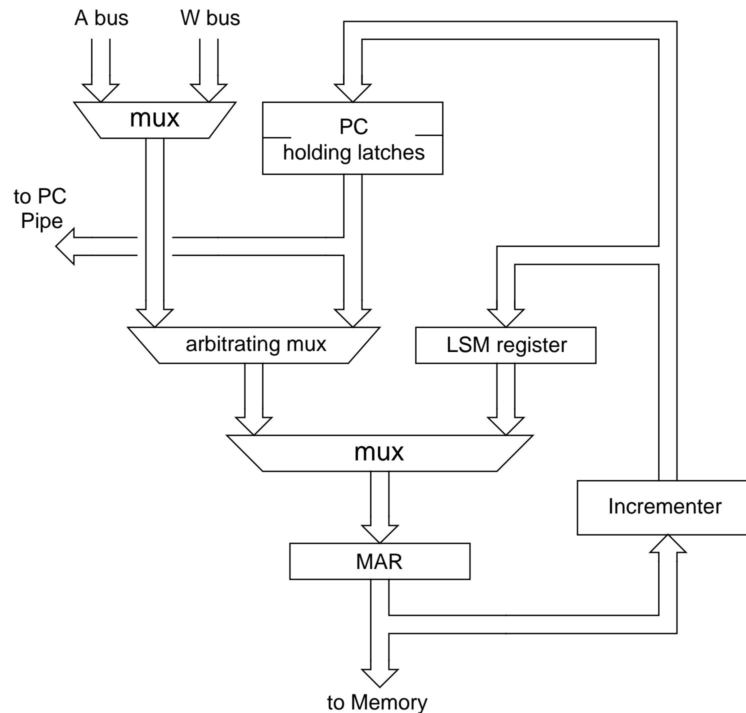


Figure 27 : Address Interface Structure.

The value of the Program Counter is available to the programmer as register R15 and can be used as a source or destination operand in the same manner as a general-purpose register. Note that writing a new value to R15, changing the PC, has the same effect as a branch instruction. However, because of the 3-stage (fetch, decode, execute) execution pipeline operation of the synchronous ARM 6, the address value read from R15 (the PC) is 8 bytes (2 instruction words) ahead of the actual address of the currently executing instruction. In order to ensure that existing ARM instruction code programs have the same functionality, some mechanism must be provided in the asynchronous implementation to mimic the behaviour when register R15 is accessed to provide instruction operands.

The **PC Pipe** is a 2-stage FIFO into which the instruction prefetch PC address value is copied after it has been used to generate a memory read access. However, after system reset the first two instruction prefetch addresses are not copied into the PC Pipe. When the first prefetched instruction eventually reaches the primary instruction decoder and synchronises with its associated R15 (PC) value, the R15 value will precede the actual memory address location of the decoded instruction by 8 bytes. As a result of the action of the PC Pipe, the asynchronous implementation can emulate the synchronous ARM behaviour of the R15 (PC) register.

The PC Pipe mechanism of maintaining the R15 value 8 bytes ahead of the currently executing instruction temporarily fails after a branch instruction executes. However, the association between R15 values and instructions is only incorrect for those instructions that do not execute, i.e. the instructions prefetched beyond the branch instruction. When the branch target instruction actually begins the decode phase, prior to execution, the PC Pipe mechanism has re-synchronised - further details can be found in [Pave94, p128].

The operation of the address interface is now described - the associated Verilog waveforms in Figures 28, 29, 30 and 31 show the instruction prefetching mechanism, a data transfer address arriving on the W bus, the address interface interaction of a LDM (LoaD Multiple) instruction and the effect of a branch address arriving on the W bus respectively:

The **instruction prefetching** cycle request (*PC_Req*), Figure 28, arrives at the address interface control arbiter along with a PC value (*PreAddr*) as an input to the Memory Address Register (MAR). Eventually, control is granted (*PCgo*) to the PC loop and the PC value is latched into the MAR by the *MAR_Req* signal. A memory read access is then initiated (*Mem_Req*) with the PC address value contained in the MAR (*MemAddr*). The control circuit then triggers the Address Incrementer (*Inc_Req*) and, once the PC value has been incremented by adding 4 (all ARM instructions are 32 bits wide and are word aligned), a completion signal (*Inc_dn*) is generated. A control signal, *PC/*

LSM, indicates whether the incrementer has been used to generate a PC value or a Load/Store Multiple (LSM) address (since a LSM instruction also uses the incrementer functionality to generate sequential addresses). The incrementer output value (*Incre*) is then latched into the first of the PC holding latches (*PC_lt1*) and, subsequently, the output of the first latch (*PCx*) is copied into the second PC holding latch (on reception of the *PC_lt2* event). The output of the second latch (the current *PC* value) is then entered into the PC Pipe (*PP_Req*) and, when the PC Pipe indicates that it has accepted the PC value (*PP_Ack*), the instruction prefetch cycle request (*PC_Req*) is again generated.

A **data transfer** address can arrive at the address interface directly from the register bank on the A bus or, in this example (Figure 29), on the ALU (write) result bus (*W_bus*). The data access request (*W_Req*) is directed to the address interface control arbiter and arbitration takes place between the data transfer request and the PC prefetch loop request. Eventually, the data transfer is given control of the address interface (*Wctl*) and a request grant signal is generated (*Wgo*). The multiplexer control signals (*MuxCtl*) are switched to allow the W bus value to pass to the input of the MAR (*Pre-Addr*), to be subsequently latched by the *MAR_Req* signal. A memory access request event is then generated (*Mem_Req*) with the address contained in the MAR (*MemAddr*). Since this is a single word transfer, the incrementer is not activated (*Inc_by*) and the data transfer is completed when an acknowledge signal is returned to the source of the W bus value (*W_Ack*).

The remaining waveforms in Figure 29 indicate a stalled PC prefetch request (*PC_Req*) which is unable to continue (*PCgo*) until the W bus acknowledge has occurred (*W_Ack*). The actual PC memory access request must also wait until the previous W bus data transfer memory cycle has completed (indicated by *Mem_Ack*). The prefetch loop then resumes by incrementing the PC access address.

For the **block data transfer** instructions (LDM/STM) involving the movement of multiple data values to or from consecutive memory locations (Figure 30), only the base address of the transfer is sent (via the A bus or, in this example, the *W_bus*) to the ad-

dress interface. The data transfer request (*W_Req*) arrives at the address interface control arbiter and again, eventually, control is given to the data transfer (*Wctl*) and the grant signal (*Wgo*) is generated. The MAR input multiplexers are switched (*MuxCtl*) and the data transfer address (*PreAddr*) is latched into the MAR (*MAR_Req*). A memory access is then initiated (*Mem_Req*) with the block data transfer base address (*MemAddr*). The address incrementer then operates (*Inc_Req*) to generate the next sequential LSM address (*Incre*). The *PC/LSM* control signal indicates that a LSM instruction triggered the address incrementer and so, when the incrementer operation has completed (*Inc_dn*), the address value is copied into the LSM (temporary storage) register (*LSM_Req*). The address interface continues to generate sequential memory addresses until a control signal (*LDM_dn*) indicates that the required number of addresses have been produced. The LSM data transfer then relinquishes control of the address interface arbiter by signalling *W_Ack*. At this point, the PC prefetching loop can again resume.

When the processor executes a **branch** instruction (Figure 31), the new PC value arrives at the address interface from the ALU via the *W_bus*. The W bus data request (*W_Req*) is directed to the address interface arbiter and eventually exclusive access is indicated (*Wctl*) and a grant signal is generated (*Wgo*). The multiplexer is again switched (*MuxCtl*) and the W bus address value is passed to the input of the MAR (*PreAddr*) where it is subsequently latched (*MAR_Req*). A memory read access is signalled (*Mem_Req*) with the new PC address contained in the MAR (*MemAddr*) to fetch the branch target instruction. The first phase of branch instruction interaction with the address interface, namely supplying the target address and initiating an instruction prefetch memory access, is now complete and control of the arbiter is released (*W_Ack*).

The second phase of the branch interaction involves restarting the PC prefetching loop with the new instruction stream addresses. Once a memory access is activated on the branch target address, the address incrementer is signalled (*Inc_Req*), the target address is incremented (*Incre*) and the incrementer completion signal is generated (*Inc_dn*). The *PC/LSM* control signal indicates that the incrementer output value is an

instruction prefetch address and it is latched through the PC holding latches (*PC_lt1* and *PC_lt2*) to become the current *PC* value. A new instruction prefetching cycle request (*PC_Req*) is directed to the address interface control arbiter and, when control is granted (*PCgo*), prefetching restarts with the new PC address value. The previous *PC_Req** request signal that was stalled at the arbiter, while the branch target address arrived on the W bus, is released (*PCgo**) when the W bus access relinquishes control of the arbiter (*W_Ack*). Control circuitry in the instruction prefetching loop is able to detect that a new PC value has arrived and so the prefetch request for the old instruction stream is discarded.

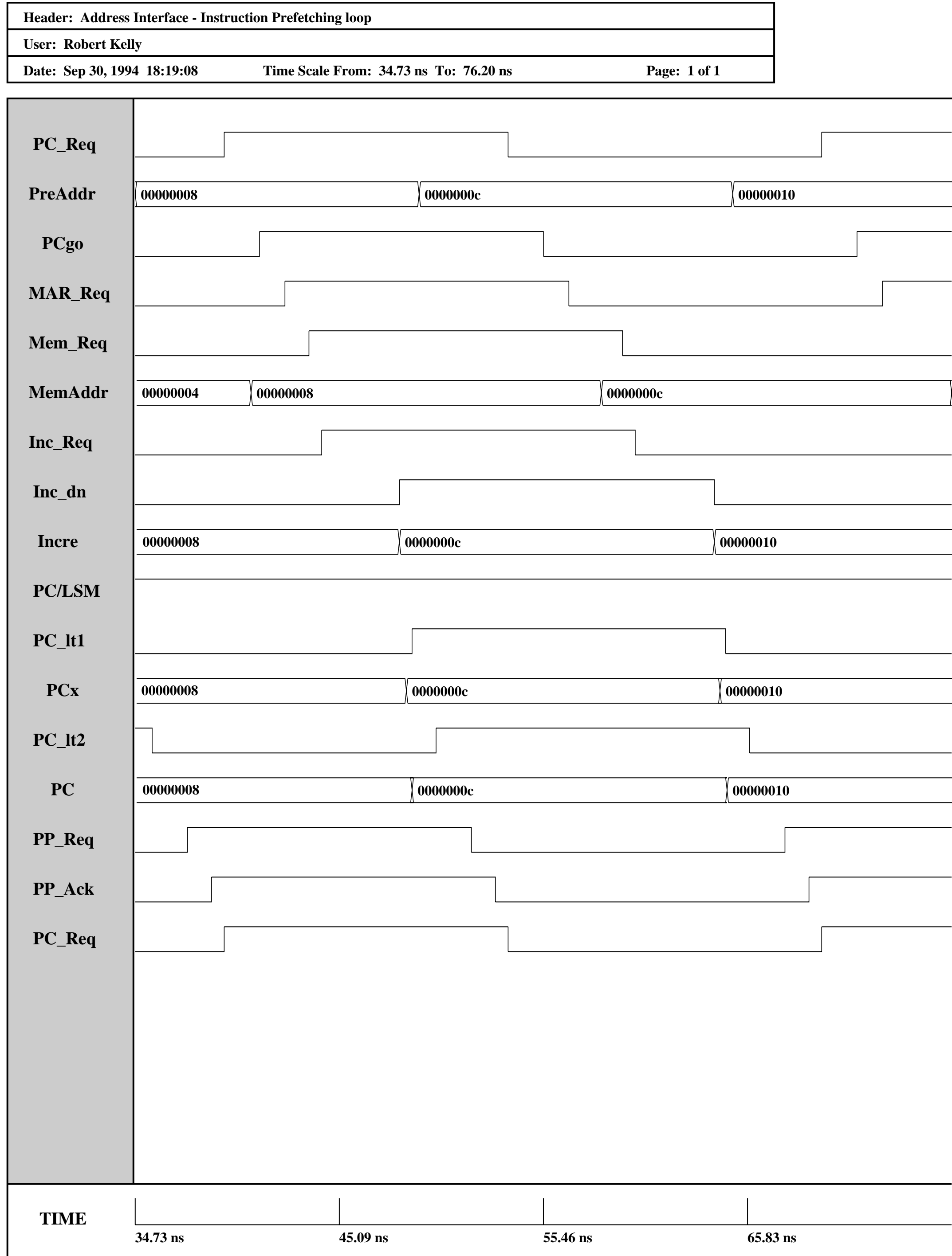


Figure 28 : Address Interface Instruction Prefetching Waveform.

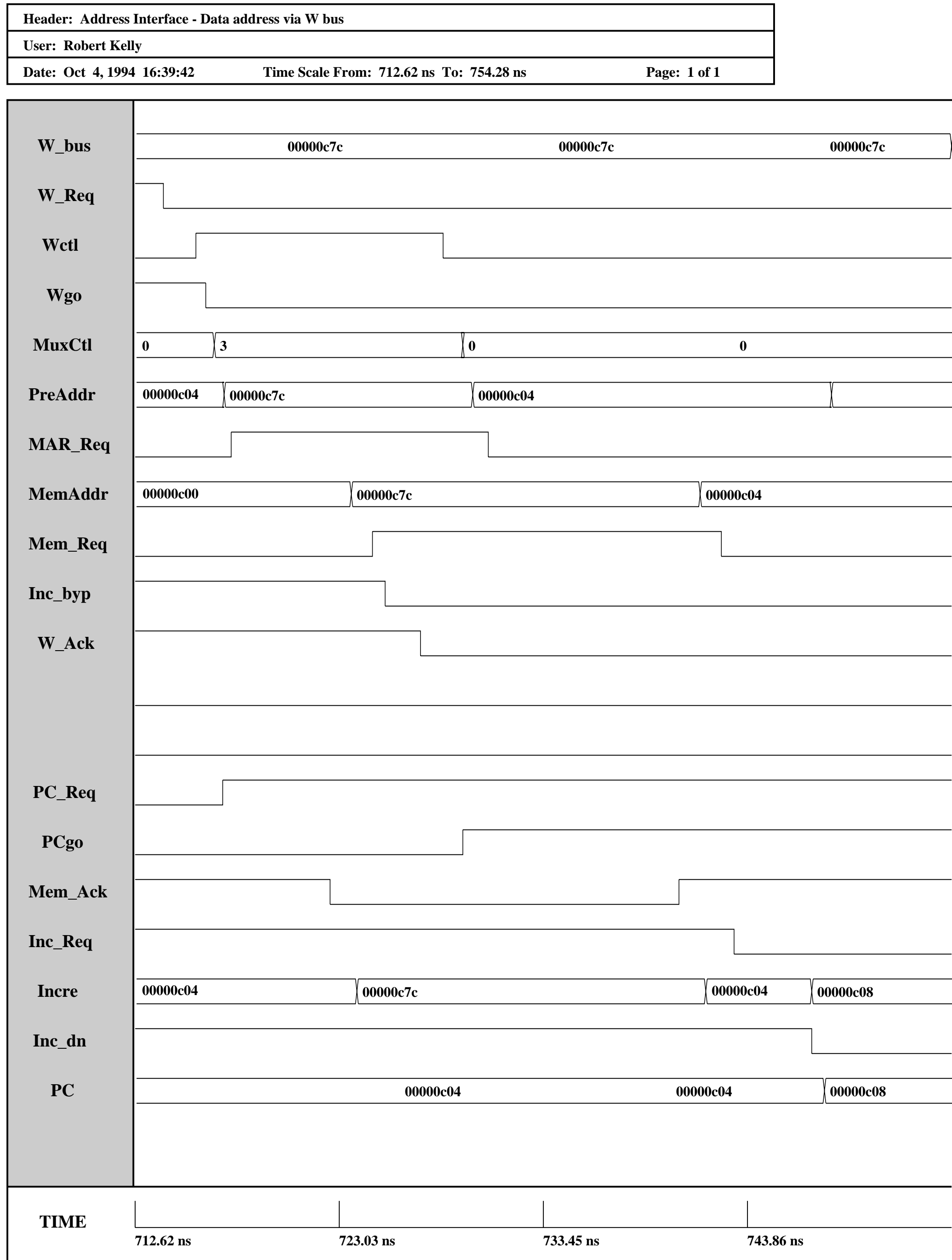


Figure 29 : Address Interface Data Transfer Waveform.

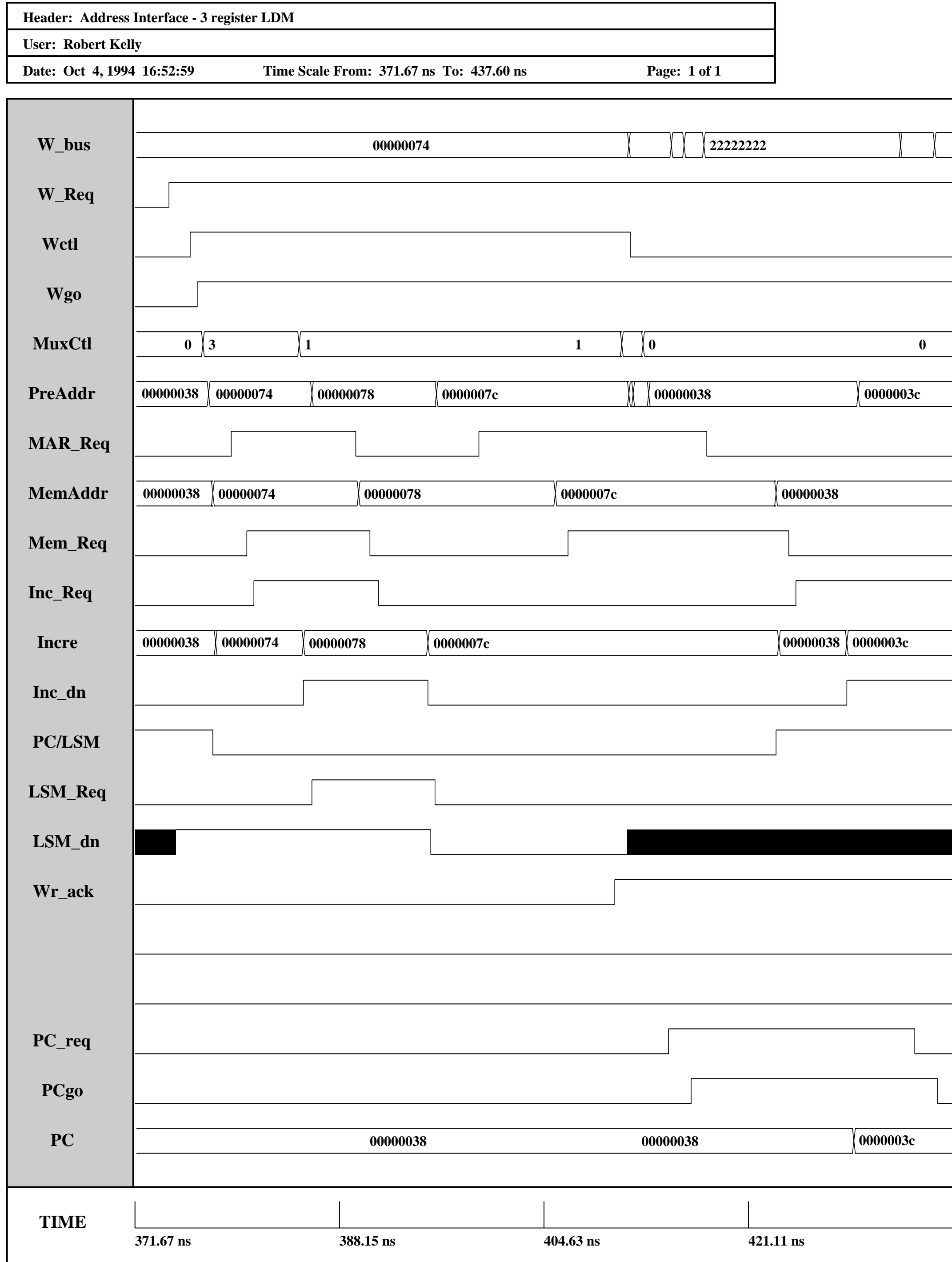


Figure 30 : Address Interface Block Data Transfer Waveform.

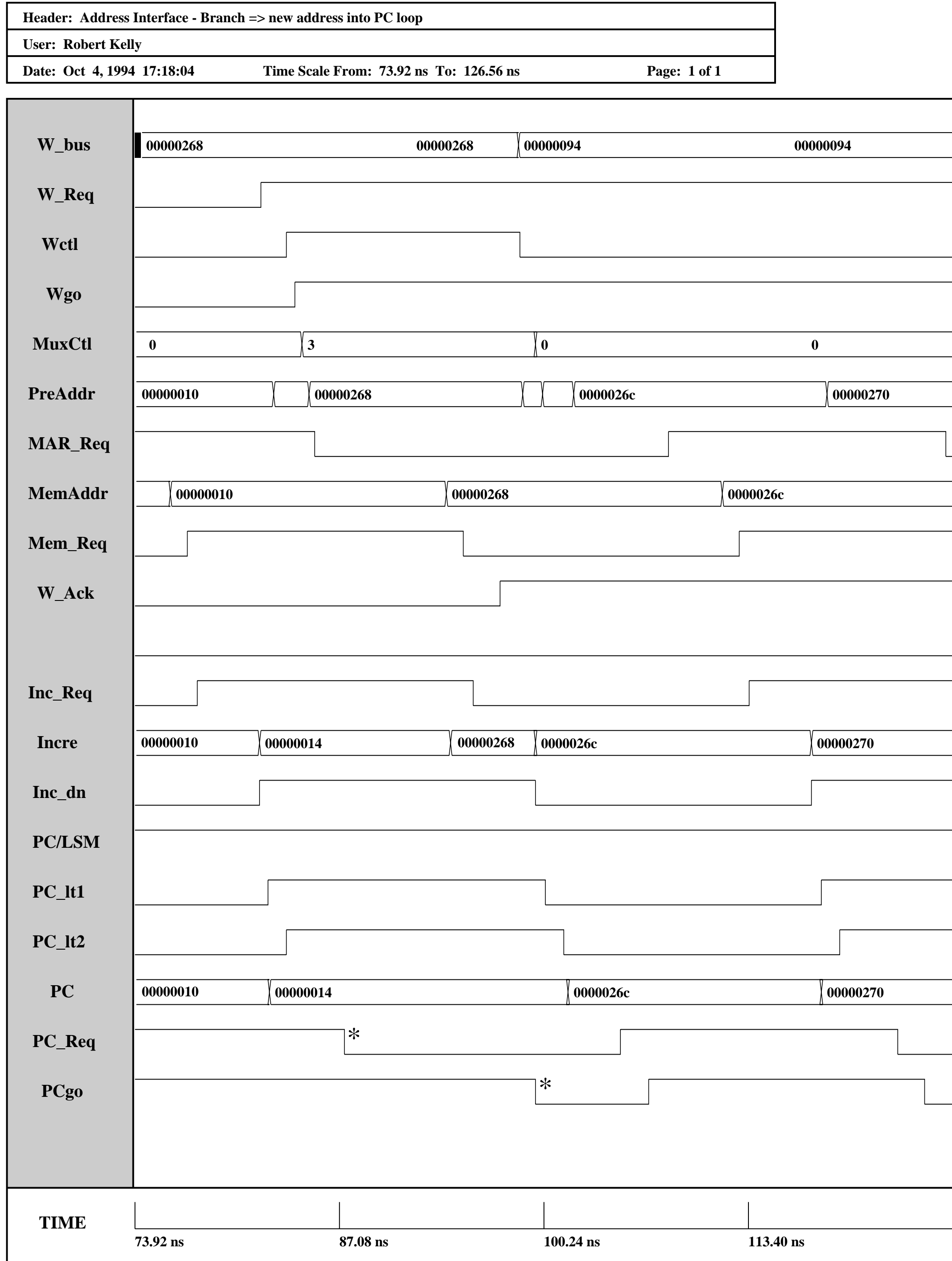


Figure 31 : Address Interface Branch Waveform.

7.2.5 Data Interface

The data interface controls the interaction between the external data bus and the processor. It handles the values returned from memory after a read access and the data values written out to memory. The overall structure of the data interface is shown below in Figure 32.

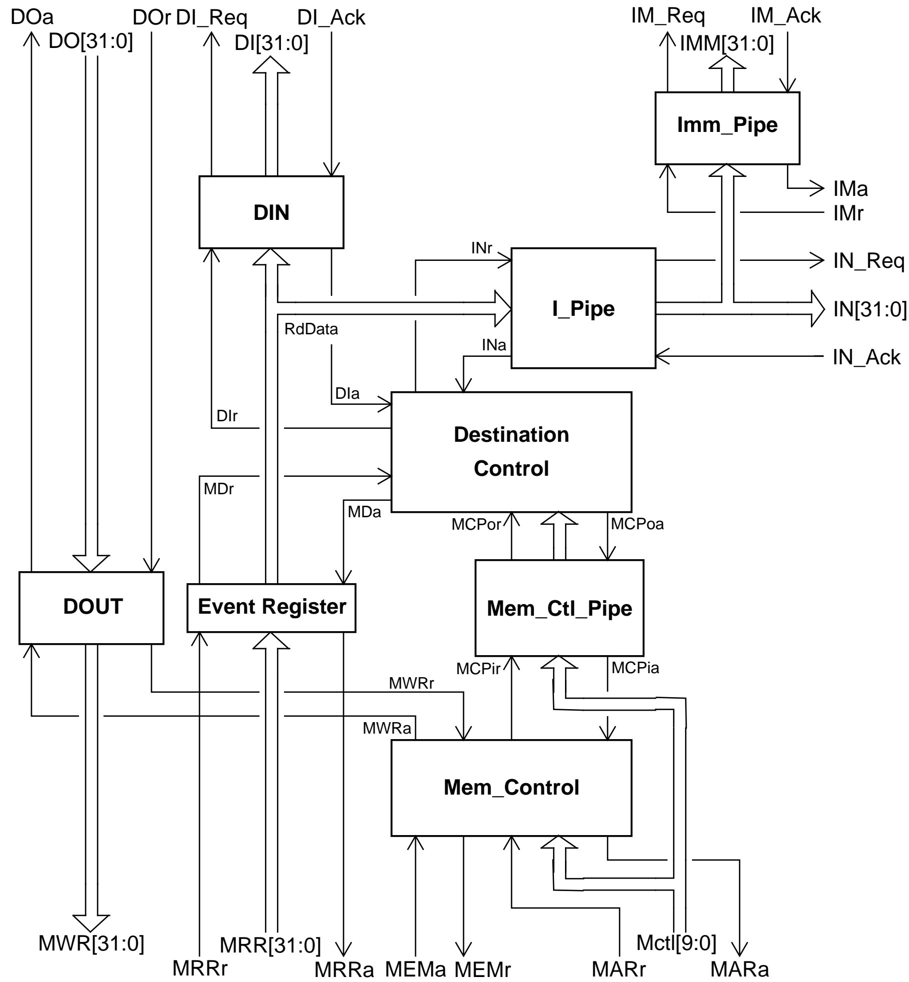


Figure 32 : Data Interface Structure.

For memory write operations, in which a byte quantity is specified, the Data Out (**DOUT**) section has the facility to replicate the least significant byte across all byte positions in the word (to enable byte writes to any byte-aligned address). The memory write data request (indicating that the data value is available) must rendezvous with the Memory Address Register request (indicating that a write address has been generated

by the Address Interface) before the external memory access request is despatched. This rendezvous occurs in the Memory Control (**Mem_Control**) section.

Once a memory read value arrives at the data interface, it is latched in an **Event Register**. The **Destination Control** block will then extract the corresponding control information from the Memory Control Pipe (**Mem_Ctl_Pipe**) for this read access. Note that the control information was entered into the Memory Control Pipe when the Address Interface generated the read access address. The retrieved control information will indicate whether the memory read value was an instruction or a data value.

Data values read from memory are passed to the Data In (**DIN**) section, where byte-rotation logic is provided to rotate values read from non-word aligned memory addresses. Also, logic exists for masking the most significant 24 bits of the data word for byte read quantities.

Incoming instructions are buffered before execution in the 5-stage Instruction FIFO Pipeline (**I_Pipe**). The I_Pipe must be 3 stages longer than the (2-stage) PC Pipe because of a complex deadlock situation - a detailed explanation can be found elsewhere [Pave94, pp130-131]. An instruction emerging from the I_Pipe may also be passed into the Immediate Field Extraction Unit (**Imm_Pipe**), so that any immediate operand can be retrieved from the appropriate fields of the instruction word prior to full decoding. The output of the Immediate Field Extraction Unit can be multiplexed onto one of the datapath operand buses, if required.

The operation of the data interface is now described - the associated Verilog waveforms in Figures 33, 34, and 35 show a data byte read operation, a data byte write operation and the reception of a prefetched instruction word from which an immediate operand is extracted respectively:

A memory **read data byte** (or word) operation (Figure 33) begins when the address interface signals (*MARr*) that a valid memory access address (*MemAddr*) has been generated. Since this is a read data transfer, the associated control information (*Mctl*) is

latched into the Memory Control Pipe (*MCPir*), while the external memory request is generated (*MEMr*). Some time later, the memory subsystem responds with a data word value (*MRR*) which is latched into the Memory Read (Event) Register by the *MRRr* request signal. An acknowledge signal is generated (*MRRa*) once the data is latched and the external memory read cycle is completed when the memory subsystem responds with the *MEMa* acknowledge event. The read input request (*MDr*), indicating that a valid read data value is contained in the Event Register (*RdData*), and the Memory Control Pipe output request (*MCPor*), signifying the availability of the associated control information for this memory access, synchronise (*Sync*) in the Destination Control section. The *Opcode* control signal indicates that the value read from memory is not an instruction and so the *DIr* request signal latches the value into the Data In (DIN) section. Additional control information is passed to the DIN block indicating that a data byte read has occurred (*B/nW*) and the position of the required byte within the word (*ByteNo*). The unwanted bytes are masked out and the byte read value is shifted into the least significant byte position (*SelByte*). An output request (*DI_Req*) is then sent to the Execution Pipeline to signal that the output of the Data In section (*DI*) is now valid and, eventually, an acknowledge event (*DI_Ack*) will be received when the byte read value has been consumed.

A **write data byte** (or word) operation (Figure 34) is initiated when a request signal (*DOr*) is received by the Data Out (DOUT) section of the data interface. It indicates that a write data word value (*DO*) has been read from the register bank and is available for transfer to memory. A control signal (*B/nW*) specifies that a byte data transfer is required. The byte-replication logic is then triggered (*Rep_Req*), which causes the least significant byte position value to be copied into all byte positions in the data word (*ByteRep*). The replicated byte value is then latched (*Rep_dn*) into the Memory Write Register (*MWR*) contained within the Data Out section. The *MWRr* request signal indicates to the Memory Control (Mem_Control) section that a write value is now ready for transfer. The *MARr* request signal indicates that the address interface has generated the associated memory address (*MemAddr*) for this write data transfer. When these two re-

quest signals synchronise (*Sync*) in the Memory Control section, a external memory data transfer is initiated (*MEMr*). Since this is a write memory access, the associated control information (*Mctl*) is not entered (no event on*MCPir*) into the Memory Control Pipe (Mem_Ctl_Pipe). A memory write access is specified (*Wen*) by the control information and, eventually, the memory subsystem responds with an acknowledge signal (*MEMa*) indicating that the memory write cycle has completed. The Memory Control section can then clear the Memory Write Register (*MWRa*) and signal the memory write cycle termination to the address interface (*MARa*).

An **instruction read** operation (Figure 35), in a similar manner to a data read operation, commences when an address interface request signal (*MARr*) arrives indicating that a PC prefetch address has been generated (*MemAddr*). The associated control information (*Mctl*) is again latched (*MCPir*) into the Memory Control Pipe before the external memory read access is requested (*MEMr*). A request signal (*MRRr*), generated by the memory subsystem, is used to latch the returned memory value (*MRR*) into the Memory Read Register. When the latch operation has completed, the data interface responds with an acknowledge signal (*MRRa*) and the external memory cycle is terminated by the *MEMa* acknowledge signal. The *MDr* signal indicating the presence of a returned memory value (*RdData*) in the Memory Read Register and the Memory Control Pipe output request (*MCPor*) synchronise in the Destination Control section. The *Opcode* control signal indicates that the memory read value is a prefetched instruction and so the value is latched (*INr*) into the Instruction Pipeline (I_Pipe). Some time later, a request signal to the primary instruction decode (*IN_Req*) indicates that the prefetched instruction (*IN*) has emerged from the Instruction Pipeline. The output of the instruction disassembler (*DIS*) shows that the instruction does indeed contain an immediate operand value. The full instruction word is subsequently latched (*IMr*) into the Immediate Field Extraction Unit (Imm_Pipe), where the immediate operand value (*IMM*) is retrieved. A request event (*IM_Req*) is sent to the Execute Unit control indicating the validity of the output of the Imm_Pipe. An acknowledge signal (*IN_Ack*) is received from the primary instruction decode stage when the instruction word has been consumed.

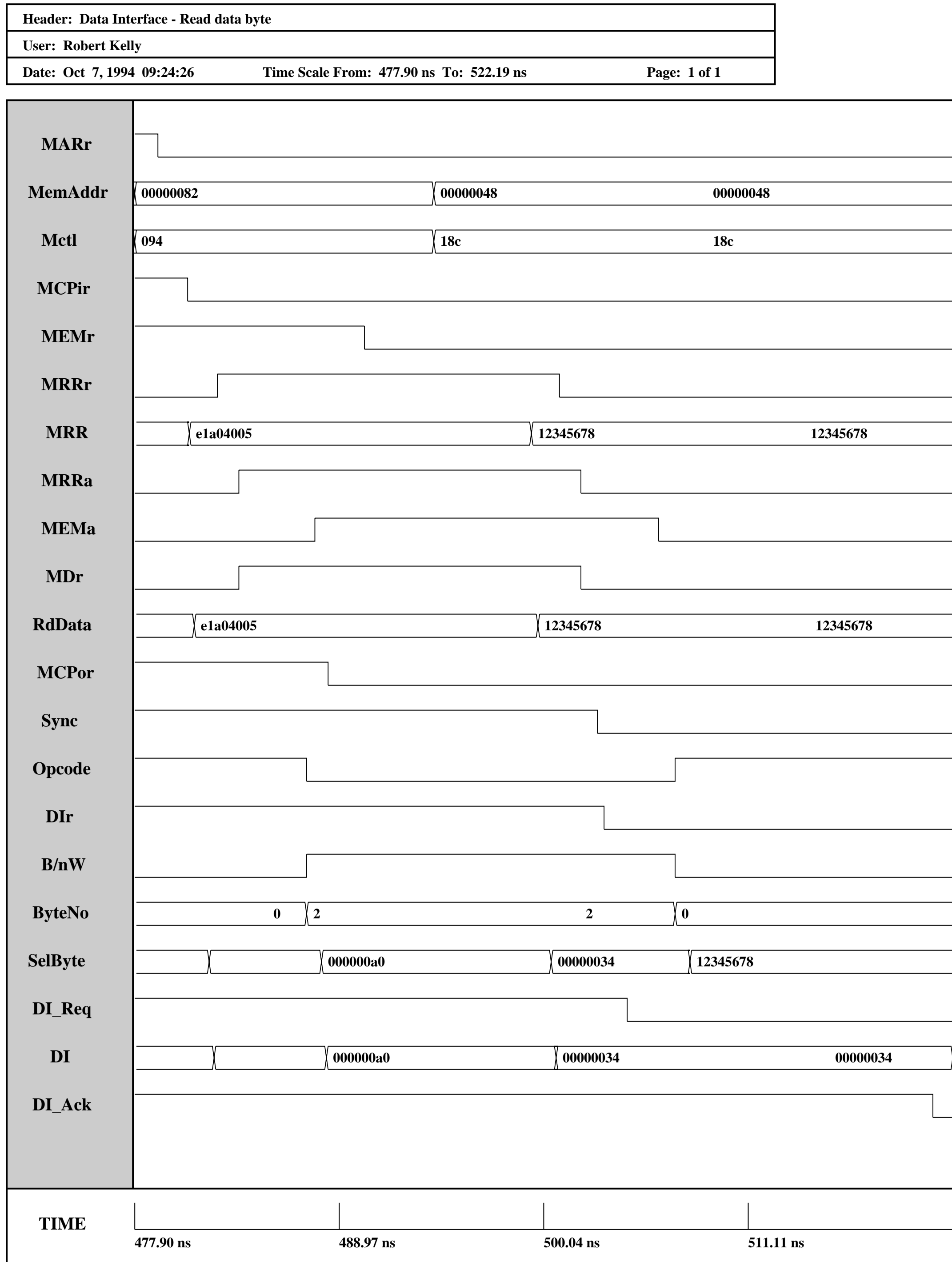


Figure 33 : Data Interface Byte Read Waveform.

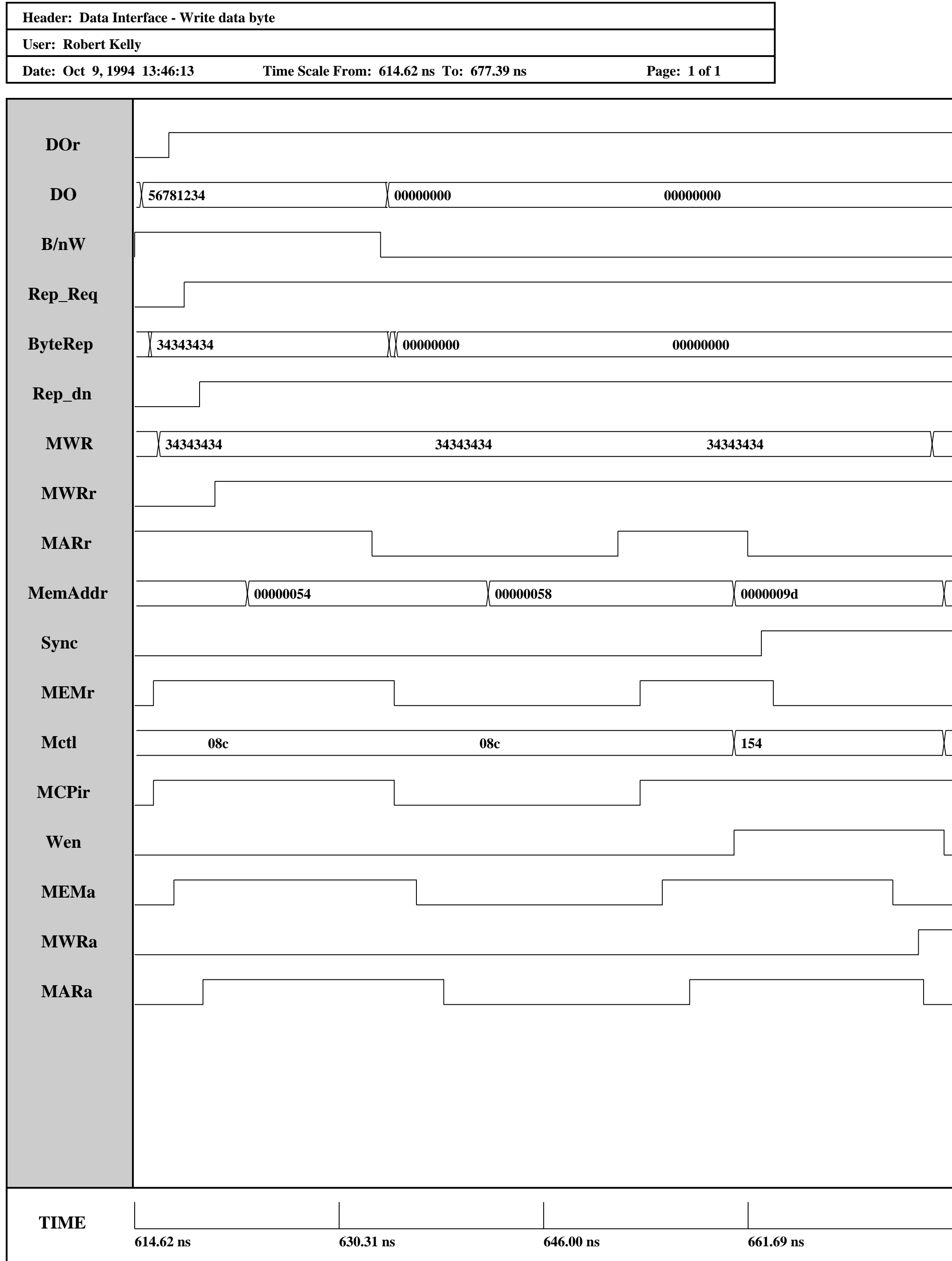


Figure 34 : Data Interface Byte Write Waveform.

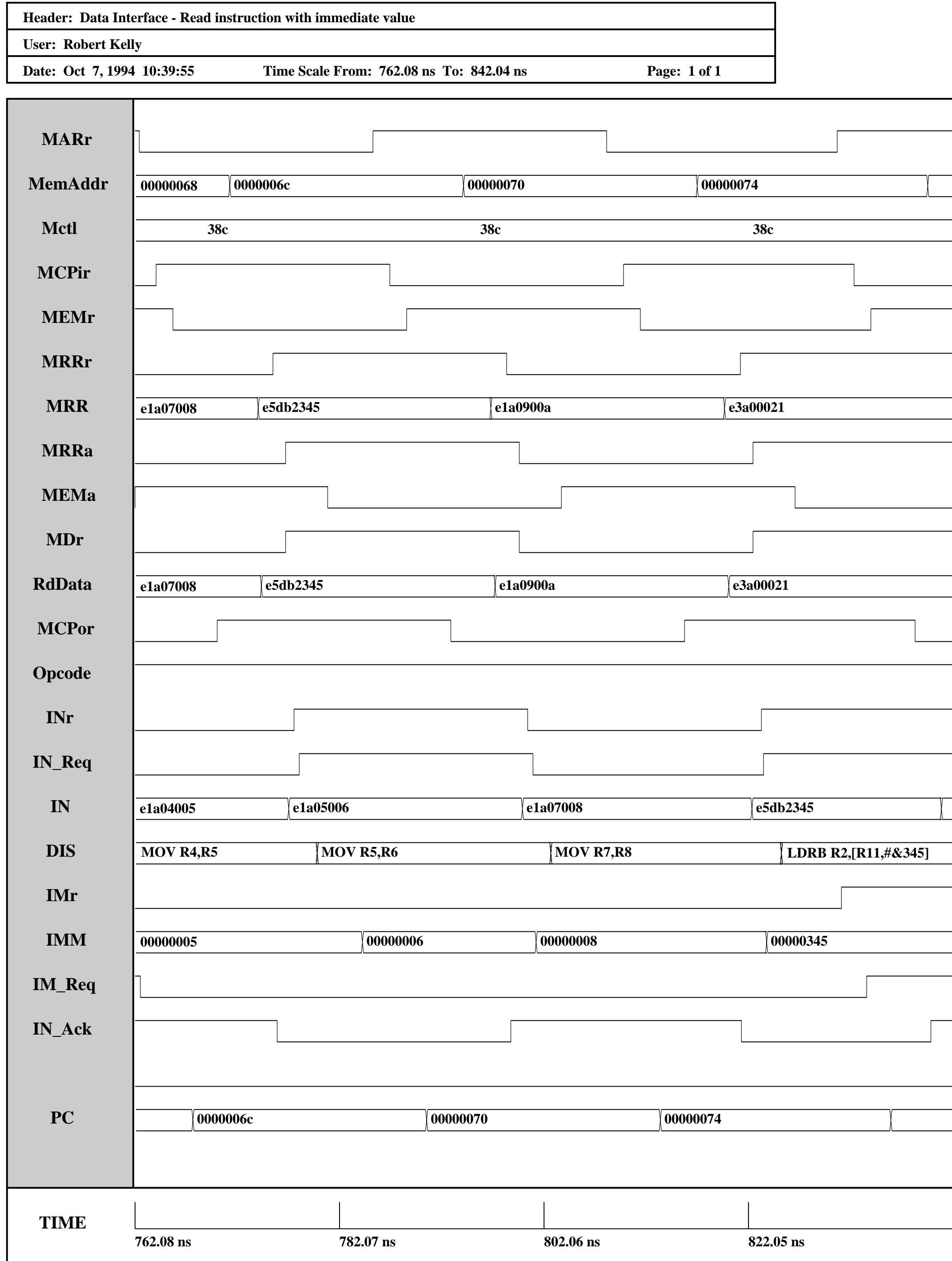


Figure 35 : Data Interface Instruction Read Waveform.

7.2.6 Execution Unit

The execution unit contains the computational logic of the processor. It comprises (see Figure 36) a Multiplier, Shifter, ALU and storage registers for the Current Program Status Register (CPSR). The Multiplier accepts two input operands to produce partial sum and partial carry outputs which are then added together in the ALU to produce a final result. It is based on an iterative shift-and-add operation using carry-save adders and incorporates early-termination detection logic.The Shifter is connected to one of the operand buses in series with the ALU allowing various shift and rotate operations to be performed on one of the ALU input values.

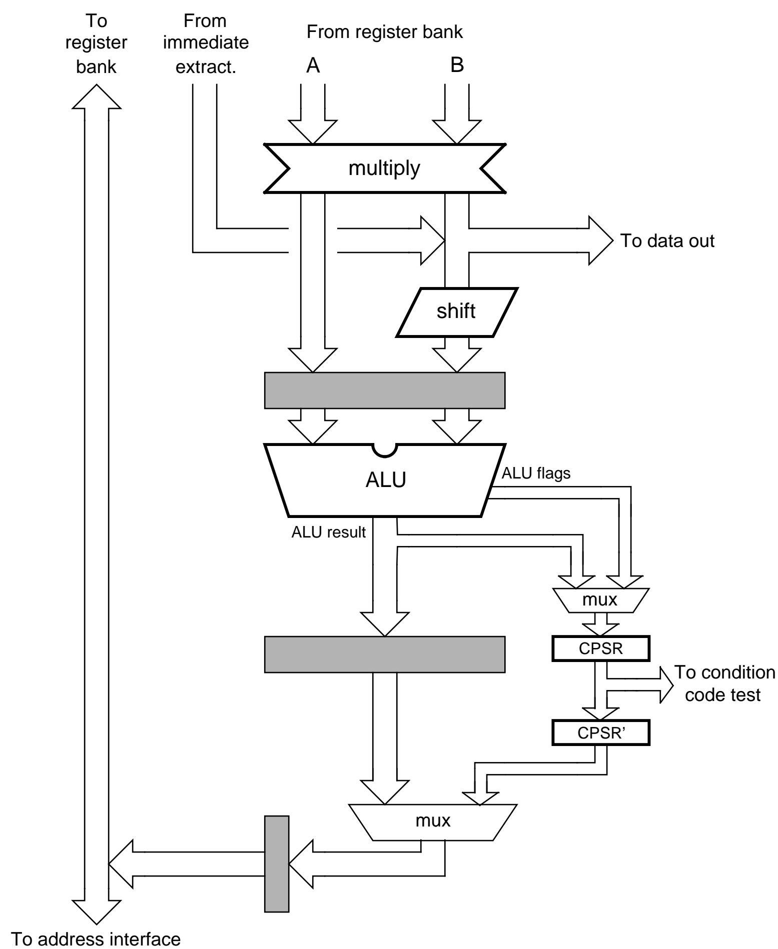


Figure 36 : Execution Unit Structure.

The Arithmetic Logic Unit (ALU) performs all the logical operations and arithmetic functions needed by the ARM architecture. The arithmetic functions requiring the ALU

to carry out an addition are potentially the most time-consuming operations because of the carry propagation between the bit positions of the calculation. A study of ARM instruction execution [Jagg90] indicates that around 20% of all instructions perform arithmetic data processing. However, since the ALU is also used in calculating addresses for data transfer and branch instructions, the actual percentage of instructions requiring an ALU addition operation is much higher than the above figure.

In a synchronous system, all ALU operations must take place within a fixed clock period and techniques, such as carry-lookahead, have been developed to reduce the time required for an addition. The ARM6 uses a carry-select mechanism. An asynchronous ALU may vary the required computation time, dependent on the actual input data values, and can determine addition operation completion by noting when carry propagation has terminated.

The operation of the MDCML Asynchronous ARM ALU has a similar high-level design to that employed in the CMOS AMULET1 [Gars93] in that addition completion is signalled when carry propagation has ceased. The actual implementation of the ALU datapath components in MDCML logic yields a much higher performance than the CMOS counterpart. However, because the circuit design technique of *wired logic* (wire-AND, wire-OR etc.) is not easily produced in MDCML technology, some aspects of the MDCML ALU control logic are slower than the equivalent CMOS circuit. In particular, the 32-bit AND function used to determine when valid signals have been asserted by all bit positions and the 32-bit NOR function used to produce the ALU output Z (zero) flag are implemented in (slow) multiple stages of 3-input gates. The average addition time in the MDCML Asynchronous ALU is much faster than the worst-case time and all logical operation are completed in a fixed (short) time period. The exploitation of data-dependent computation time results in a simple ALU design of comparable performance to existing synchronous designs which incorporate carry-lookahead or carry-select techniques.

The operation of the execution unit is now described - the associated Verilog wave-forms in Figures 37 and 38 show a multiply operation (followed by the addition of the partial sum and partial carry multiplier outputs in the ALU) and an ALU logical operation involving a shift of one of the ALU input operands respectively:

The execution of a **multiply** instruction (Figure 37) begins when the datapath input request (*RD_Req*) arrives indicating that the two operands specified by the instruction have been read from the register bank (*A_bus* and *B_bus*). A control signal (*Mult*) shows that a multiply operation is required and so the multiplier request signal (*Mul_-Req*) is generated. When the multiply operation has finished two outputs are produced, the partial sum (*Psum*) and the partial carry (*Pcarry*), and a completion signal (*Mul_dn*). The partial sum and carry are then latched into the ALU input operand event register by the *Op_Req* signal. At this point the register bank output register is no longer required to hold the initial instruction operands stable and the execution unit indicates this by generating an acknowledge signal (*RD_Ack*). The two ALU input operands (*A_op* and *B_op*) are subjected to the required ALU function (*Afunc*) - in this case, an addition to combine the partial sum and carry - when the ALU is enabled (*ALU_Enb*). When the ALU operation has terminated, an output value (*ALU*) is produced along with a completion signal (*ALU_dn*). The ALU output latch is then closed (*ALU_lt*), holding the ALU output result (*Result*) stable. A request event is generated (*O_Req*) to indicate that the result value is available for copying into the execution unit output register (not shown in Figure 36). A W (write) bus request signal (*W_Req*) is forwarded to the Write Control unit, while the execution unit output register value (*W_bus*) is placed on the W bus. The Write Control unit will 'steer' the W_Req request signal to the appropriate function unit based on the associated control information (*Wctl*) for this instruction. Eventually, the specified function unit (in this example, the register bank) will respond with an acknowledge signal (*W_Ack*) when the result value has been received.

For the execution of an instruction involving a (register) **shifted operand** (Figure 38), the input request (*RD_Req*) from the register bank again indicates the validity of

the input operands (*A_bus* and *B_bus*). The shifter is enabled (*Sh_Enb*) with the appropriate function control signals (*Sfunc*) and eventually the shifter output result (*Shift*) is produced along with a completion signal (*Sh_dn*). The A bus instruction operand and the shifted B bus operand are then latched into the ALU input operand event register by the *Op_Req* signal and the register bank input request is subsequently acknowledged (*RD_Ack*). From this point on the execution unit control signals and sequence of events is similar to the ALU operation described for the multiply instruction previously. The ALU input operands (*A_op* and *B_op*) are again subjected to the required function (*Afunc* - in this example, an AND operation) when the ALU is enabled (*ALU_Enb*) and, eventually, an output value (*ALU*) is produced followed by a completion signal (*ALU_dn*). The ALU output value is latched (*ALU_lt*) into the ALU result latch (*Result*) and the execution unit output register is signalled (*O_Req*). The W bus request signal (*W_Req*) is generated when the execution unit output value (*W_bus*) is valid and the appropriate function unit is signalled based on the associated instruction result control information (*Wctl*). An acknowledge signal (*W_Ack*) is received when the result destination function unit has consumed the value.

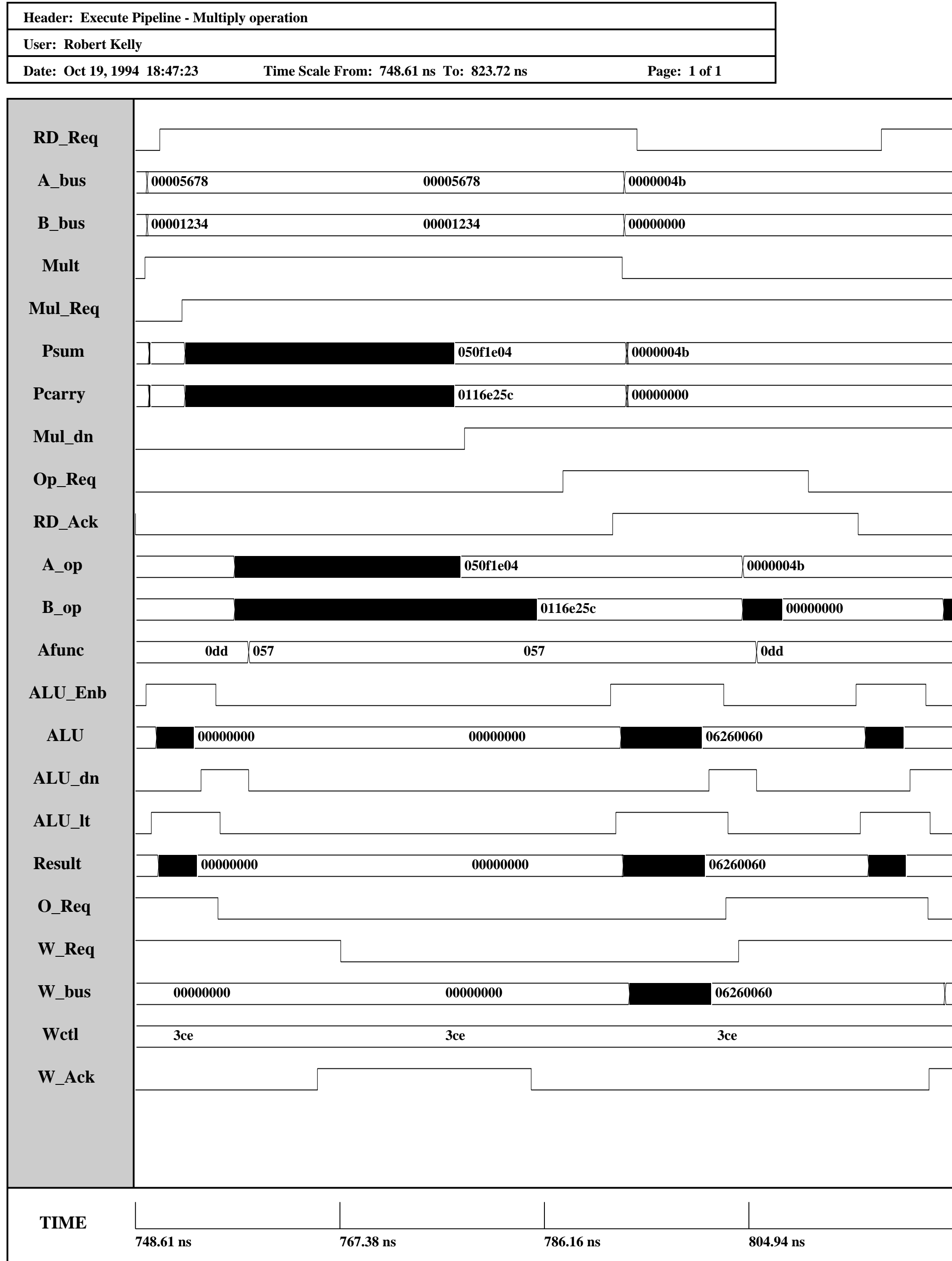


Figure 37 : Execution Unit Multiply Waveform.

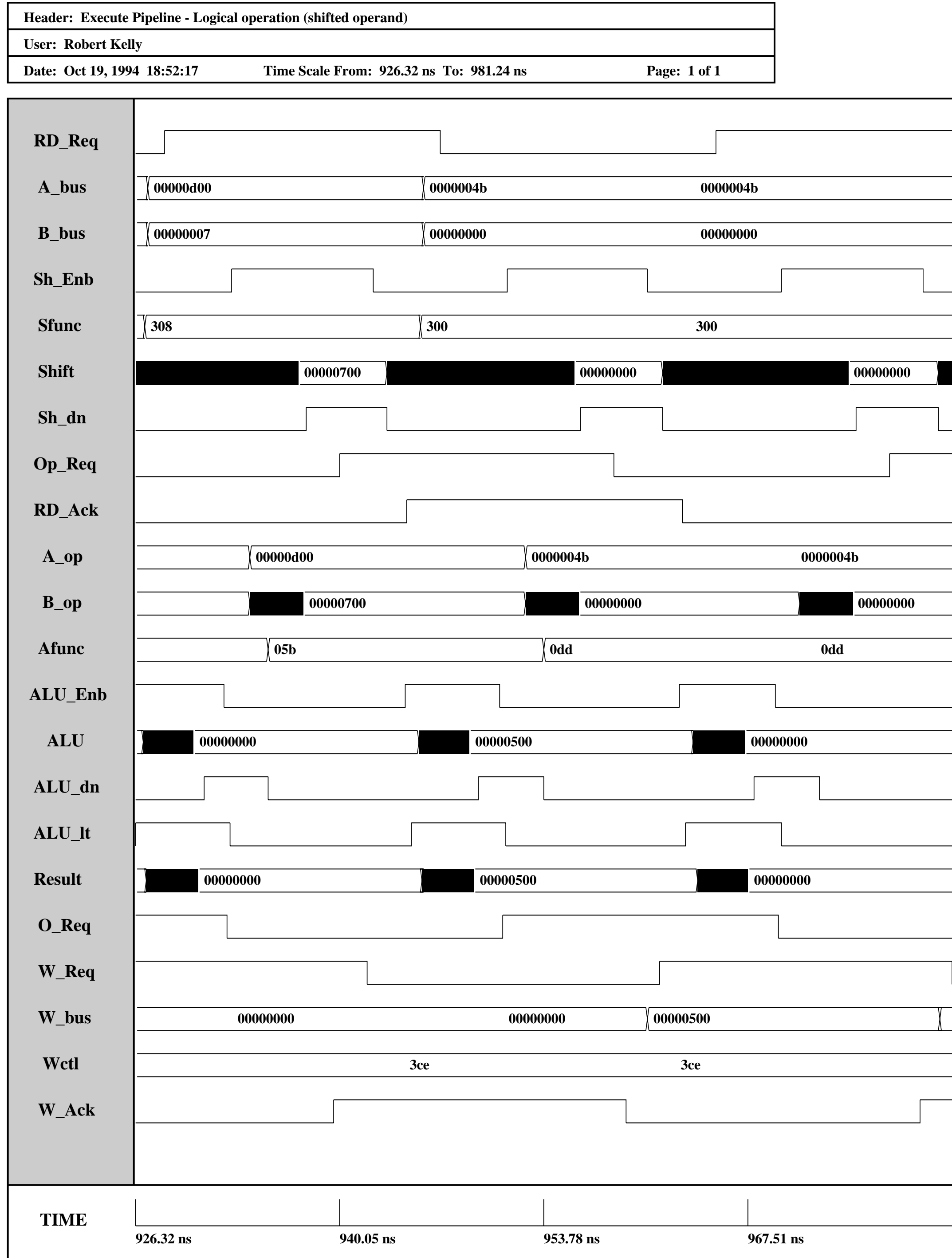


Figure 38 : Execution Unit Shifted ALU Operand Waveform.

7.2.7 Comments on the MDCML Asynchronous ARM Design

The MDCML ARM demonstrates that the implementation of a simple RISC architecture using asynchronous design techniques is attainable. The complex design task is made manageable by employing a modular design methodology, namely **micropipelines**, with subsystems communicating via a well-defined protocol i.e. the **transition-signalling bundled-data interface**. The design of the Register Bank control logic, with the novel, arbiter-free method of allowing concurrent read and write operation interaction, gives an example of how new design problems can be overcome. The data-dependent operation of the ALU shows how an asynchronous system can take advantage of the variable processing rates of a particular functional unit in order to increase overall performance. Also, the autonomous action of the instruction prefetching mechanism in the Address Interface demonstrates the independent operation of the component subsystems.

The MDCML Asynchronous ARM exhibits a very high degree of concurrency which is suggested in many of the Verilog waveforms shown earlier in the chapter. This is as a result of the self-timed constituent function units operation being solely dependent on input data availability. As a consequence of this asynchronous computational parallelism, the total system state at any particular instant is difficult to determine. Similarly, the effects of the interactions between two communicating subsystems, in an overall system context, are difficult to quantify. Developing an understanding of the total system operation is still in the early stages, and the design changes required to increase overall system performance are not immediately obvious. The production of a realistic simulation model of the entire system (described in the following chapter) which has the ability to execute real ARM instruction code programs has proved invaluable in exploring the complex behaviour of the running system.

8. Architectural Modelling

8.1 Introduction

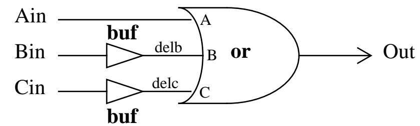
Verilog is an industry-standard Hardware Description Language which is integrated into the CAD system supplied by the MDCML technology provider, GEC-Plessey Semiconductors (GPS). The entire process, therefore, of architectural modelling through schematic design capture and physical layout to bipolar technology fabrication is more easily accomplished.

Architectural modelling of a system seeks to hide the lowest levels of the implementation complexity from the conceptual design, so that alternative design ideas can be more easily considered and evaluated. The design process iteratively refines the higher levels of abstraction to move towards an implementation of the prototype system. At each stage in the process, the Verilog system model can be simulated to provide an indication of the design correctness and system performance.

8.2 Modelling

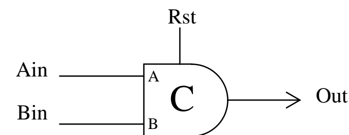
The initial requirement in developing a model of a prototype system is the production of a library of components that can be used to construct larger functional subsystems. The Verilog HDL has a range of logic primitives incorporated into the language but, because of the switching characteristics of the different signal levels in MDCML, the standard primitives must be combined to produce models of the MDCML gate-level equivalents (see Section 6.3). For example a 3-input OR gate can be modelled in the following manner:

```
‘timescale 1ps/1ps
module or3 (Out,   Ain,Bin,Cin);

‘define  A_del   230
‘define  B_del   300
‘define  C_del   400

output  Out;
input   Ain,Bin,Cin;
wire     delb,delc;

buf   #(‘B_del - ‘A_del)  g1 (delb,  Bin);
buf   #(‘C_del - ‘A_del)  g2 (delc,  Cin);
or    #(‘A_del)           g3 (Out,  Ain,delb,delc);

endmodule
```



An ‘asynchronous control element’ library is also produced using the behavioural modelling language constructs of Verilog. This comprises the micropipeline control circuit elements outlined in Section 3.1. The Verilog behavioural model of the Muller-C element is shown below:

```
‘timescale 1ps/1ps
module MullC (Out,   Ain,Bin,Rst);

‘define  A_del     470
‘define  B_del     640
‘define  Rst_del   370

output  Out;
reg      Out;
input   Ain,Bin,Rst;

always @ (Ain)
      if ((!Rst) && ((Ain===Bin) || (Ain===‘bx)))
            #(‘A_del)   Out = Ain;

always @ (Bin)
      if ((!Rst) && ((Ain===Bin) || (Bin===‘bx)))
            #(‘B_del)   Out = Bin;

always @ (Rst)
      case (Rst)
            1‘b1:      #(‘Rst_del)   Out = 0;
            1‘b0:      if (Ain===Bin)
                              #(‘B_del)   Out = Bin;
            1‘bx:      #(‘Rst_del)   Out = ‘bx;
      endcase

endmodule
```



The dynamic simulation behaviour of the above Muller-C element is provided by the 3 concurrently-executing **always @** statements, one for each of the inputs: *Ain*, *Bin* and *Rst*.

As an illustration of the operation of the model, the *Ain* behavioural code is explained:

> At every change of the *Ain* signal (**always** @ (Ain)), if the *Rst* (reset) signal is inactive (!Rst) and both inputs have the same value (Ain === Bin) or the *Ain* input is undefined (Ain === ‘bx), then the *Ain* input signal is passed to the output (Out = Ain) after the appropriate delay (#‘A_del).

Note that any undefined input signal arriving at the Muller-C is propagated to the output. This feature assists in the detection of incorrect operation (see Section 8.5.3).

Larger components, such as 32-bit Event Registers (Section 3.2.1) can be constructed from their constituent elements: a Muller-C gate and 32 Capture-Pass latches. However, since Event Registers are widely used throughout the MDCML Asynchronous ARM, a behavioural model of an Event Register is produced which improves simulator performance. That is, a single model is invoked for any input data signal change rather than multiple invocations of the constituent models. Also, by producing a single behavioural model for a larger function, additional checking can be incorporated into the model structure to report all occurrences of incorrect circuit operation. For example, the reception of two successive input request events, without an intervening input acknowledge event, results in an error message being displayed during the simulation execution.

The complex computational subsystems of the Asynchronous ARM architecture, including the ALU, shifter and multiplier, are also modelled as behavioural modules. It is much easier to handle the input and output bus values of such components as single data entities (e.g. 32-bit integers) rather than manipulating the individual bit values. For example, consider adding two 32-bit operands in the ALU:

```
input   [31:0] A, B;
output  [31:0] out;
reg     [31:0] out;

        out = A + B;
```

Once the bit-widths of the input and output buses are specified, the addition result assignment to the output bus is achieved by means of a single arithmetic operator.

The complete MDCML Asynchronous ARM model consists of a single module which instantiates the component subsystems to produce a hierarchical composition of asynchronous modules. The autonomous subsystems communicate using the two-phase bundled-data interface, and the total system is self-starting from reset. Once the global reset signal is deactivated, instruction prefetching commences (from the external memory model) leading to execution of the test program instructions.

The processor diagram shown overleaf in Figure 39 shows the major functional subsystems and the significant control signal and data bundles connected between them. To assist in the clarity of the diagram, some of the signals found in the Verilog processor ‘core’ model in Figure 40 have not been included in Figure 39. The signal names in the bolder typeface in the processor diagram indicate the connections to the external environment. At the top right-hand corner of the diagram are the bundled-data interface signals used to communicate with the external memory system. These include the Memory Access Control Information, Memory Address, Write Data and Read Data values and the associated protocol control signals. The memory subsystem is modelled using the Verilog behavioural language to generate the required data values and the communication protocol control signal sequences. The two signals names at the bottom of the figure (*nAbt* and *Dabt[1:0]*) handle the fault responses of the memory system. The ‘normal’ and ‘fast’ interrupt signals (*Nirq* and *Nfiq*) are shown at the top left-hand corner.

The Verilog model in Figure 40, illustrates the top-level components of the MDCML Asynchronous ARM and the connectivity of the processor signals. The full hierarchical model developed by the author is given in Appendix A. For example, the Register Bank (**Reg**) has the instantiation name *rg*; it produces the *RGa, RWa* and *RDr* output signals along with two 32-bit output buses (*Na[31:0]* and *Nb[31:0]*). The Register Bank has five input signals (*RGr, RWr*, *Wc[2]*, *Wsel* and *RDa*), in addition to the global reset signal (*Rst*), and has three input buses: a 32-bit Write (result) bus, a 30-bit PC (program counter) bus and a 28-bit control bus (*Rs[27:0]*).

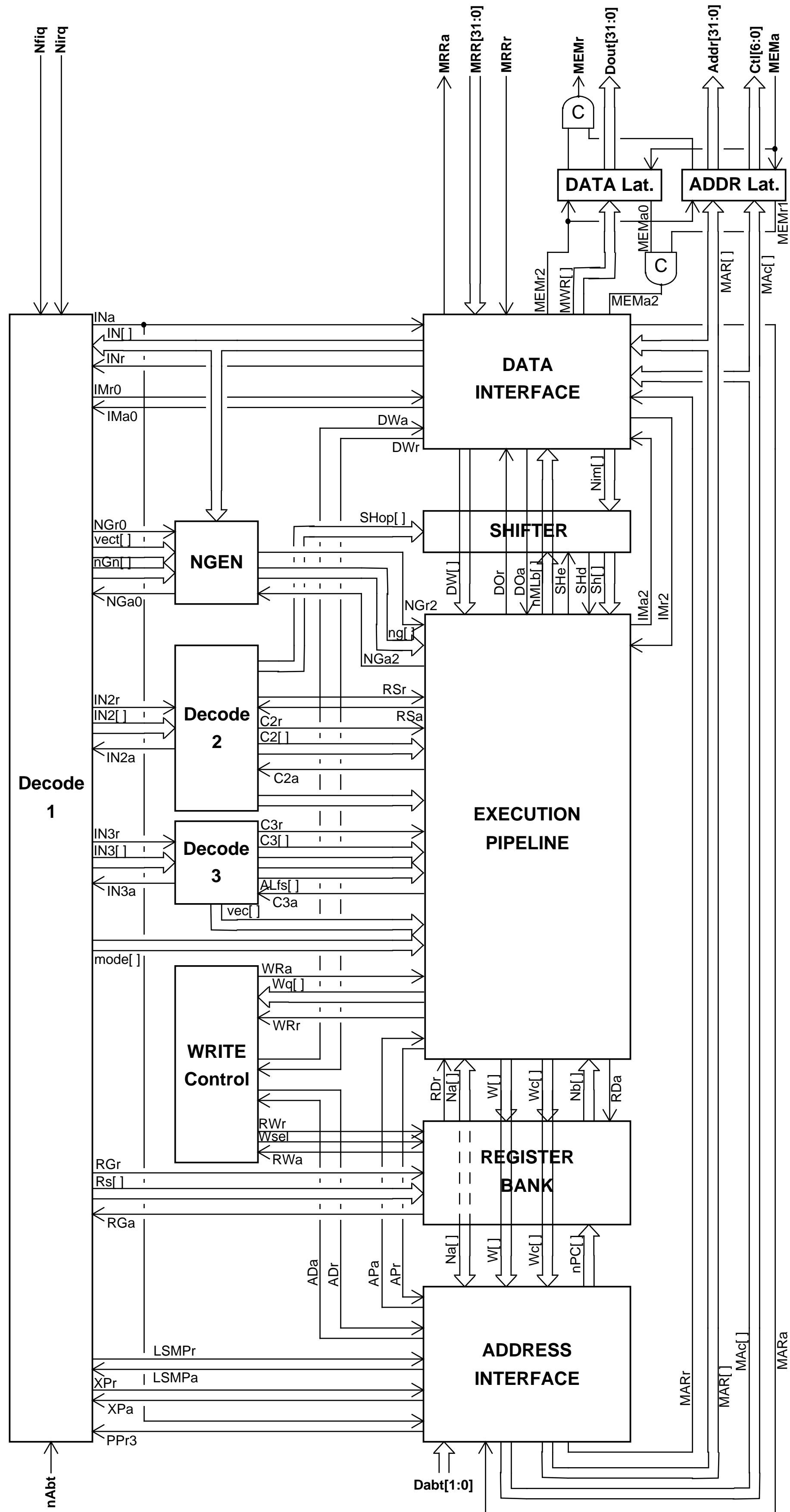


Figure 39 : MDCML Asynchronous ARM Processor Diagram.

```
//                           MDCML Micropipelined ARM
`timescale 10ps /10ps

module ARMstCore (Add[31:0],Dout[31:0],Ctl[6:0],MEMr,MRRa,
                       MRR[31:0],Nfiq,Nirq,Dabt[1:0],PAbt,MEMa,MRRr,BigEnd,nAbt,Rst);
`include "ARMstCore.inc"

// Ctl[] = Seq, Inc, Ren, Wen, Usr, B/W, Opc

// memory data in blocks and instruction pipeline
DatInt dat (MARa,DOa,MEMr2,MWR[31:0],MRRa,DWr,DW[31:0],DWusr,
            DWv,DWpc,INr,IN[31:0],flo[1:0],Nim[31:0],IMa0,IMr2,
                        MARr,DOr,nMLb[31:0],DObw,MEMa2,MRRr,MRR[31:0],PAbt,DWa,
                        INa,{MAR[1:0],MAc[4:0],Mval,MdPC,MpPC},IMr0,IMa2,BigEnd,Rst);

EvtReg #32 lat0 (Dout[31:0],MEMa0,MEMr0,    MWR[31:0],MEMr2,MEMa,Rst);

Cgate2     c0   (MEMa2,    MEMa0,MEMa1,Rst);
Cgate2     c1   (MEMr,     MEMr0,MEMr1,Rst);

// 1st decode stage
Decode1 dec1 (RGr,Rs[27:0],IN2r,IN2[25:0],IN3r,IN3[19:0],PCsel,XPr,
              XLa,INa,IMr0,LSMPr,nTRM,r15,NGr0,nGn[1:0],vect[2:0],
                            INr,IN[31:0],flo[1:0],Nfiq,Nirq,PCpar,RGa,IN2a,IN3a,XPa,
                            XLr,PPr3,IMa0,LSMPa,ALUgo,ALUok,mode[5:0],NGa0,nAbt,Rst);

// 1st execution and 2nd decode stage
Reg rg (RGa,RWa,RDr,Na[31:0],Nb[31:0],
                              RGr,Rs[27:0],nPC[31:2],RWr,W[31:0],Wc[2],Wsel,RDa,Rst);

NGen nGen (NGa0,NGr2,ng[5:0],    NGr0,IN[15:0],nGn[1:0],vect[2:0],NGa2,Rst);

Decode2 dec2 (IN2a,RSa,C2r,{Imd[6:0],SHop[9:0],DObw,c2[7:0]},
                                                IN2r,IN2[25:0],RSr,Na[7:0],C2a,Rst);

// IN[]  = Xt[1:0],PCpar,cond[3:0],sctls[2:0],I[11:5],
//                                           DObw,toRs,cpCP,~toDO,~toA,nGen,~Mult,NImm
// Imd[] = Xt[1:0],PCpar,cond[3:0]
// c2[]  = toRs,cpCP,~toDO,~toA,nGen,~Mult,NImm

// 3rd decode stage
Decode3 dec3 (IN3a,C3r,ALfs[9:0],{vec3[2:0],c3[22:0]},    IN3r,IN3[19:0],C3a,Rst);

// c3[]  = UseCP,S,F,C,Wcp[2:0],Ral,Rcnd,~ALUwt,~DabtWt,
//                              tPCp[1:0],Wreg,Wadd,SP,LSM,Ren,Wen,B/W,Opc,destPC,Rsel

// 3rd control and execution stages
Shift shft (Sh[31:0],ShC,SHd,    nMLb[31:0],Nim[31:0],c2[0],SHop[9:0],psrC,SHe);

ExecP excP (RDa,C2a,C3a,NGa2,SHe,IMa2,PCpar,ALUgo,ALUok,mode[5:0],psrC,
            WRr,W[31:0],Wc[9:0],Wq[1:0],APr,DOr,nMLb[31:0],RSr,Dabt0,
                            RDr,Na[31:0],Nb[31:0],C2r,c2[7:0],C3r,c3[22:0],vec3[2:0],
                            ALfs[9:0],NGr2,ng[5:0],Sh[31:0],ShC,SHd,IMr2,Imd[6:0],
                            DW[31:0],DWv,DWusr,WRa,Wsel,APa,DOa,RSa,Dabt[1:0],Rst);

// write bus control
WrCtl wctl (DWa,WRa,RWr,ADr,Wsel,    DWr,DWpc,DWv,WRr,Wq[1:0],RWa,ADa,WLx,Rst);

// the memory address interface
AddInt add (ADa,WLx,APa,PPr3,XPa,XLr,nPC[31:2],LSMPa,
            MARr,MAR[31:0],{MAc[6:0],Mval,MdPC,MpPC},
                                      ADr,W[31:0],Wc[9:0],APr,Na[31:0],LSMPr,nTRM,
                                      r15,INa,XPr,XLa,PCsel,MARa,Dabt[1],Dabt0,Rst);

// MAc[] = Seq, Inc, Ren, Wen, Usr, B/W, Opc
// Mval  = valid    MdPC = destPC    MpPC = PCpar

EvtReg2 #(32,7) lat1 (Add[31:0],Ctl[6:0],MEMa1,MEMr1,
                                                MAR[31:0],MAc[6:0],MEMr2,MEMa,Rst);

endmodule // ARMstCore
```

Figure 40 : MDCML Asynchronous ARM ‘Top-Level’ Verilog Model.

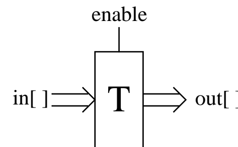
The Verilog comment at line 8 of Figure 40 (*// Ctl[] = Seq,Inc,Ren,Wen,Usr,B/W,Opc*) indicates the component signals that comprise the *Ctl[6:0]* external memory control bus. Some of the other MDCML Asynchronous ARM control buses are also expanded in the comment lines.

8.3 Features

8.3.1 Instantiation Parameters

One useful feature of the Verilog modelling environment is the use of parameters when instantiating components. These parameters may be used, for example, to specify different propagation delay times for different instances of the same module (to reflect particular gate loading effects) or to specify multiple bit-widths for certain components. To illustrate this point, a register can be modelled by specifying a multiple bit-width parameter for the data input and output nets of a latch.

```
‘timescale 1ps/1ps
module T_latch (out, in, enable);
parameter width=1;                    // default data width = 1
parameter Data_delay=330;
parameter Enb_delay=490;

output [width-1:0] out;
reg      [width-1:0] out;
input   [width-1:0] in;
input enable;

always @ (enable)
      case (enable)
          1‘b1:   #‘Enb_delay out = in;
          1‘bx:   out = ‘bx;
      endcase

always @ (in)
      if (enable)
           #‘Data_delay out = in;

endmodule
```



When instantiating components, the required parameters must be specified in the same order as they are given in the particular component definition - in the case of the T_latch above: width,Data_delay,Enb_delay. If no parameters are specified, the default values are used, i.e. width =1, Data_delay = 330ps, Enb_delay = 490ps.

The modules are then instantiated in the following manner:

Two single-bit latches with differing data propagation delays

```
T_latch #(1,300,500)  t1   (out1, in1, enb1);
T_latch #(1,400,500)  t2   (out2, in2, enb2);
```

t1 is a single-bit T_latch, with a 300ps input-output data propagation delay and a 500ps enable-output propagation delay, where the data input signal is called *in1*, the output is called *out1* and the enable signal is called *enb1*.

t2 is also a single-bit T_latch, this time with a 400ps input-output data propagation delay and again an enable-output propagation delay of 500ps.

A 3-stage pipeline for 32-bit data values:

```
T_latch #(32,300,500)  p1  (o1[31:0], pin[31:0], enb);
T_latch #(32,300,500)  p2  (o2[31:0], o1[31:0], Nenb);
T_latch #(32,300,500)  p3  (pout[31:0], o2[31:0], enb);
```

The pipeline is constructed by instantiating T_latch components with 32-bit data widths. The input of the pipeline, *pin[31:0]*, is fed into the input of the first latch, p1. The output of p1, *o1[31:0]*, is fed into the input of the second latch, p2, and so on.

The enable signals of the successive stages of the pipeline operate in antiphase, causing data values to move one stage along the pipeline for every two transitions of the enable signal.

8.3.2 Test Vector Generation

A standard technique of generating test patterns for validating a fabricated chip is to apply stimuli to the simulation model of the design and then dump the values of the significant control signals and data buses at suitable time intervals to an *activity* file. In a synchronous system, this normally occurs at the clock edge, when all signals are usually stable. For an asynchronous system, however, given that subsystems operate concurrently at their own rate, the sequential ordering of changes in logic level of two independent signals internal to two separate subsystems cannot be specified. Therefore, the total system state at any given instant cannot be known.

One method used to automatically generate test patterns for a micropipelined system is to locally delay the acknowledge signal into each subsystem until sufficient time has elapsed so that all internal signals have reached a stable state. Effectively, the module is deadlocked awaiting the acknowledge input. The subsystem state is then recorded at the instant of the acknowledge input event using the **$fstrobe()** Verilog function. Test patterns for the entire chip can be produced by delaying the external memory access acknowledge input for each memory access and dumping the control and bus values of interest.

The **$strobe()** Verilog system task allows the value, at the end of the current timestep, of any signal wire or register to be displayed on the standard output device. The **$fstrobe()** function allows the values to be written to a file via an output channel identifier. For example:

```
always @ (input)
        $fstrobe(chan_id, " %b %b %h", input, output, state);
```

The **$fstrobe()** task is triggered on every input signal change (**always** @ (*input*)). The signal values are written to the file which was bound to the *chan_id* channel identifier when it was initially opened. The signal values are written on the same line, for each input signal change, in the following order: *input*, *output*, *state*. The format of the signal values (" %b %b %h") is Binary for the *input* and *output*, and Hexadecimal for the *state*.

The example illustrated overleaf is of the MDCML Asynchronous ARM Chip model (ARMst), which consists of the processor core and the bond pad driver circuits. In order to reduce the pin count, the input data bus (MRR[31:0]) and the output data bus (Dout[31:0]) use the same external data bus (Xd[31:0]) by means of tristate driver circuitry.

The activity file is opened, in a Verilog **initial** timing control block, using the following file operation system task:

```
dump_chan = $fopen("ARMst_vecs");
if (dump_chan == 0) $finish;            // quit simulation if $fopen() fails.
```

The interface signals for the ARMst module are then recorded in the activity file by the $fstrobe() system task whenever the Reset (XRst), Memory Read Request (XRr) or Memory Access Acknowledge (XMa) signals change:

```
always @(XRst or XRr or XMa)
    $fstrobe(dump_chan,
        " %b %b %b %b %b %b %b %b %b %b %h | %h %b %b %b  -  %0d",
            XBigEnd, XnAbt, XPAbt, XDabt, XNfiq, XNirq, Xdbe,
            XRr, XMa, XRst, Xd, Xa, Xc, XMr, XRa, $time);
```

The format of the resulting activity file is shown below:

```
         Test vectors for ARMst chip (core + peripherals)

               Inputs                             Outputs
  X                                 |
  B                                 |
  i X X  X X X                      |
  g n P  D N N X     X              |
  E A A  a f i d X X R              |                    X X
  n b b  b i r b R M s     X        |     X       X      M R
  d t t  t q q e r a t     d        |     a       c      r a     time
--*-*-*-10-*-*-*-*-*-*-********--|--********-*******-*-*-----****-
  0 0 x xx 1 1 x 0 x 1 xxxxxxxx  |  xxxxxxxx xxxxxxx x x  -  0
  0 0 x 00 1 1 x 0 0 1 xxxxxxxx  |  xxxxxxxx xxxxxxx x x  -  101
  0 0 0 00 1 1 0 0 0 0 xxxxxxxx  |  00000000 0110001 0 0  -  1000
  0 0 0 00 1 1 0 1 0 0 ea000098  |  00000000 0110001 1 0  -  2736
  0 0 0 00 1 1 0 1 1 0 ea000098  |  00000000 0110001 1 1  -  3271
  0 0 0 00 1 1 0 0 1 0 ea000022  |  00000004 1110001 0 1  -  4514
  0 0 0 00 1 1 0 0 0 0 ea000022  |  00000004 1110001 0 0  -  5049
  0 0 0 00 1 1 0 1 0 0 ea00008e  |  00000008 1110001 1 0  -  6292
  0 0 0 00 1 1 0 1 1 0 ea00008e  |  00000008 1110001 1 1  -  6827
```

The prototype silicon can then be tested by subjecting the test specimen to the input stimulus given on the left-hand side of each line in the activity file. Eventually, the specimen outputs should (for a fully-functioning device) assume the associated test vector file output values for each stimulus line.

Since the signal values are only ‘sampled’ when the system state is stable, there is a risk that timing errors may be overlooked. However, only timing errors on the external interface signals may be missed, since any internal data-bundling timing errors will propagate incorrect data values to the outputs - which will then be detected. Design effort must be directed to the external interface control elements to ensure data-bundling errors are eliminated.

8.4 Code Execution

8.4.1 Compilation Method

The executable binary is generated from the actual program and an ARM assembler file which contains various initialisation and library functions. The files are compiled, assembled and linked using the ARM Cross-Development Toolkit. This allows code to be generated by a SPARC-based workstation for execution on an ARM processor. A binary executable is produced, which is then converted to a text format suitable for loading into the asynchronous ARM Verilog model.

8.4.2 Validation Suite

Since the MDCML Asynchronous ARM is binary code-compatible with the existing synchronous ARM devices, the test program suite used by Advanced RISC Machines (ARM) Ltd. to test prototype devices can also be used to test the design of the asynchronous implementation.

The ARM Validation Suite consists of over a dozen test programs written in ARM assembler [Cock87]. The suite includes programs to exercise the data processing subsystems of the ARM architecture, involving the Arithmetic Logic Unit, Shifter and Multiplier. Further validation programs test the operation of the Register Bank, including the reading and writing of the Current and Saved Processor Status Registers (CPSR and SPSRs) and the interaction of the processor with the external memory system via the Load/Store Register (LDR/STR) and Load/Store Multiple (LDM/STM) instructions. The branch (and branch-and-link) mechanism of the processor is also fully tested.

As mentioned previously, the MDCML Asynchronous ARM has no support for coprocessor interaction and the Multiply-with-Accumulate (MLA) instruction is not implemented, therefore these aspects of the ARM Validation Suite are not considered during the design test phase.

The simulated execution of the ARM Validation programs revealed a number of errors in the Asynchronous ARM model. In particular, running the Multiplier function test program contained in the Validation Suite exposed an error in the Verilog Multiplier module. The cause of the problem was traced to a specific feature of the Verilog modelling environment. When the assignment of a new value to a bus or control signal occurs, but the newly assigned value is the same as the previous value, then no Verilog **event** is generated for the assignment. This means that any event control statement dependent on the signal value (eg. **always** @ (signal)) is not triggered. The Multiplier behavioural model had to be modified and the addition of an extra control signal was required.

Complete verification of the MDCML Asynchronous ARM architectural model, by running the ARM Validation Suite, gives a significant degree of confidence in the overall asynchronous design and the component subsystems.

8.4.3 Dhrystone Benchmark

As a high-level language platform, a computer architecture should efficiently execute those features of a programming language that are most frequently used in actual programs. This ability can be measured by a program known as a *benchmark*. A benchmark can be a real application program supplied with specific input data chosen to provide a representative task or a specially-written (synthetic) program incorporating a wide range of high-level language statements and constructs.

The original Dhrystone synthetic benchmark program (written in Ada) was published in the CACM in October 1984 [Weic84]. A ‘C’ version was produced in 1988. The program contains statements of a high-level programming language in a distribution which is considered representative of a general-purpose, integer-computational processor workload. The program statement statistics used to develop the Dhrystone benchmark are based on the execution of over 700 programs written in several languages.

The actual benchmark statement distribution is as follows (‘C’ version):

assignments	51.0%
control statements	32.3%
procedure, function calls	16.7%

The distribution of statements is also balanced with respect to operators (arithmetic, logical, comparison etc.), operand type (integer, character, pointer, Boolean etc.) and operand locality (global, local, procedure parameters, function results etc.). The program does not compute anything meaningful, but is syntactically and semantically correct.

There are several areas where the execution details (compiler influence, timing measurement method, cache interaction etc.) have to be checked very carefully whenever a synthetic benchmark program is used for comparison of different processors or different systems. However, for evaluation of design alternatives of the functional components of a prototype microprocessor, the Dhrystone benchmark, with its representative mix of program statement types, provides a useful metric.

The executable binary is generated from three files: two ‘C’ source files (dhry_1.c and dhry_2.c) which contain the actual benchmark program and an ARM assembler file which contains initialisation and library functions. A 16Kbyte binary executable is produced, which is then converted to the text format suitable for loading into the Verilog external memory model.

The model executes 1 Dhrystone loop in approximately 344 seconds and indicates a simulated time of 22.9 microseconds, the ratio of the actual running time to the simulated time is 15,000,000:1. This translates to a Dhrystone benchmark figure of around 43,500 Dhrystones per second. For the purposes of the benchmark execution, an external memory access time figure of 5ns is assumed. Also, the result is based on typical parameters for the underlying 1.2μm bipolar technology.

For comparison, the 1μm CMOS AMULET1 device yields a figure of 20,500 Dhrystones per second [Furb94b].

8.5 Usage

8.5.1 Instrumentation

Since the Verilog language has a rich and powerful behavioural modelling capability, custom design tools and system modelling instrumentation functions can also be quickly and easily produced. The following tools, which assist the digital engineer in exploring the dynamic behaviour of the prototype system, have been written by the author.

The data bundling constraint (see Section 2.2.4) is an integral and necessary part of the interface protocol. A **Bundle Checker** module has been written in the Verilog behavioural modelling language and attached to each of the data "bundles" of interest (data bus + request signal) to determine the validity of the data value change and request signal event sequencing. This has enabled modules with an insufficient bundling tolerance to be identified and modified. The bundle checking code is also incorporated into the behavioural representation of the Event Register module, since these components are widely used throughout the Asynchronous ARM design.

Usually, a **1 nanosecond** bundling margin is considered safe, i.e. the data arrives at least 1ns before the request event. However, an Event Register has a 'built-in' bundling margin of around 1.2ns because of the circuit topology (see Figure 6). The ReqIN request signal must pass through the Muller-C element and then through a power driver circuit (not shown in Figure 6) before the Capture-Pass element begins to latch the incoming data, Din. Therefore, even if the data and request signals arrive simultaneously at the Event Register external inputs, the 'built-in' bundling margin results in a safe transfer of data.

A sample output of the Bundle Checker module and Event Register during a simulation run is shown below (all times are given in picoseconds):

```
Bund_Chk  test.sys.cpu.core.rg.LkF.ChkA: Margin = 90 @ 14573
Bund_Chk  test.sys.cpu.core.rg.LkF.ChkM: Margin = 90 @ 26342

EvtReg    test.sys.cpu.core.dat.memCP.l0:    2730 @ 9671
EvtReg    test.sys.cpu.core.dec1.l3:         4230 @ 35185
Bund_Chk  test.sys.cpu.core.rg.ChkI:         2450 @ 6206
EvtReg    test.sys.cpu.core.rg.wlat:         1730 @ 35891
Bund_Chk  test.sys.cpu.core.rg.LkF.ChkA:       90 @ 14573
Bund_Chk  test.sys.cpu.core.rg.LkF.ChkM:       90 @ 26342
EvtReg    test.sys.cpu.core.add.xpipe.e1:   21280 @ 46342
```

The first block shows where (and when) the Bundle Checker has detected a bundling margin below 1000ps (i.e. 1ns) while the simulation is running. The second block involves each bundle checking component (including Event Registers) reporting its minimum bundling values at the end of the simulation run. Note that some of the modules indicate a bundling margin well in excess of 1ns. This suggests areas where the control circuit performance may be increased.

Another, behaviourally-modelled, design tool which has been implemented is the **Pipeline Occupancy Monitor**. This is used to collect information regarding the effectiveness of each of the FIFO buffering pipelines used throughout the design, and can clearly be used to influence the pipeline depth in the design. The effect that the number of pipeline stages has on performance is examined in greater detail in Section 8.6.4.

A further useful tool when attempting to understand the operation of a microprocessor is a disassembler, since it is often useful to know the specific instruction that a particular functional unit is processing. This can be achieved by disassembling the 32-bit value representing the instruction (as in most RISC architectures, the ARM instructions are a fixed width). A **Verilog Disassembler** module can be connected to the input stage of the instruction (buffering) pipeline in the Data Interface, to note when a particular instruction of interest is (pre)fetched from external memory. Alternatively, it could be connected to the input of the Instruction Decoder to determine when the instruction actually begins decoding. Usually, the latter option is chosen because it represents the commencement of the actual instruction execution.

Writing the disassembler module in the Verilog behavioural language was relatively straightforward for the author because of two factors: The Verilog language syntax is very similar to an existing high-level procedural language ('C') of which the author has programming experience; and the instruction set generally follows the RISC principle of having few instruction formats with regular bit-field positions. The output of the Verilog-ARM Disassembler is in the form of a text string which is suitable for display in conjunction with other signal and bus values using the Verilog waveform display described in the following section.

A Disassembler output example can be seen towards the bottom of Figure 35 (labelled **DIS**) in the previous chapter. In this case, the disassembler module is connected to the input stage of the Primary Instruction Decoder.

8.5.2 Graphical Output

The Verilog waveform output mechanism is implemented by the **gr_waves()** system task. The user can continuously monitor the waveforms via the interactive graphics interface as the simulation progresses. Two different screens are provided: the *Waves* screen, on which the signal waveforms are displayed as timing diagrams, and the *Select* screen, which displays the list of signals from which the user can choose a subset for current display. The unknown (or X) state of a signal or bus is displayed as a solid filled box. The high impedance (or Z) state is displayed as a horizontal line which is vertically centred between the '0' and '1' levels. The *gr_waves* system task was used to produce the waveform diagrams illustrated in the previous chapter.

Verilog provides an interactive graphics interface to display data as a screen of text along with the formatted values of system model nets and registers. The **gr_regs()** system task defines the layout of the screen and specifies the text and variables to be displayed and the appropriate formats. The graphics screen is updated whenever a value changes for any of the variables defined in the *gr_regs* task during the simulation execution.

A time bar is displayed at the top of the *gr_regs* window representing the total time period of the simulation execution. The interactive mode allows the user to select a particular instant in time by positioning the cursor at a particular point on the time bar - the required data values for the corresponding time are then displayed in the *gr_regs* window.

REGS

RDRW | LINE | LINE | PAGE | PAGE | t = 20893.49 ns

ASYNCHRONOUS ARM REGISTER BANK

	Current	Previous	Mod-time
R0	ffffffff	00000031	2073830
R1	ffffffff	32000000	2068817
R2	00000031	00003120	2036578
R3	00000032	00003220	2038547
R4	0000075c	0000075c	2079613
R5	0000077c	00000001	1552493
R6	00000003	00000002	1652119
R7	00000041	00000001	1650150
R8	00000003	00003e90	1560896
R9	00003e9c	00000000	1279084
R10	00000000	00000000	0
R11	00000748	000007cc	1550574
R12	01010101	0000075c	1715231
R13	00000728	00000720	2077934
R14	00010000	4051464e	1874060
R15	00000000	00000000	0
R16	00000000	00000000	0
R17	00000000	00000000	0
R18	00000000	00000000	0
R19	00000000	00000000	0
R20	00000000	00000000	0
R21	00000000	00000000	0
R22	00000000	00000000	0
R23	00000000	00000000	0
R24	00000bd0	00000a74	23420
R25	00000000	00000000	0
R26	00000000	00000000	0
R27	00000000	00000000	0
R28	00000000	00000000	0
R29	00000000	00000000	0
R30	00001534	00001530	2089349
R31	00000000	00000000	0
R32	00000000	00000000	0
R33	00000000	00000000	0
R34	00000000	00000000	0
R35	00000000	00000000	0
R36	00000000	00000000	0

Figure 41 : Register Bank Display using Verilog gr_regs() system task.

This graphical output feature is particularly useful for displaying information about the internal state of the prototype system. The Asynchronous ARM Register Bank is displayed using this feature in Figure 41. The Register Bank consists of 31 general-purpose registers (including the Program Counter (PC) - R30) and 6 SPSRs (Saved Processor Status Registers). Only a subset of the entire Register Bank is 'visible' to the programmer in any one of the processor execution modes. The *gr_regs* Register Bank

display shows the current simulation time in the top right-hand corner (t = 20893.49ns). The value of each of the 37 registers, at this particular time, is shown in the left-hand column. The previous value of each of the registers, and the (simulation) time at which each register was modified is shown in the middle and right-hand columns respectively.

Verilog also provides the capability to view data as dynamically changing bar graphs. The **gr_bars**() system task allows the user to set up charts with multiple bar graphs and update the bars as the simulation proceeds.

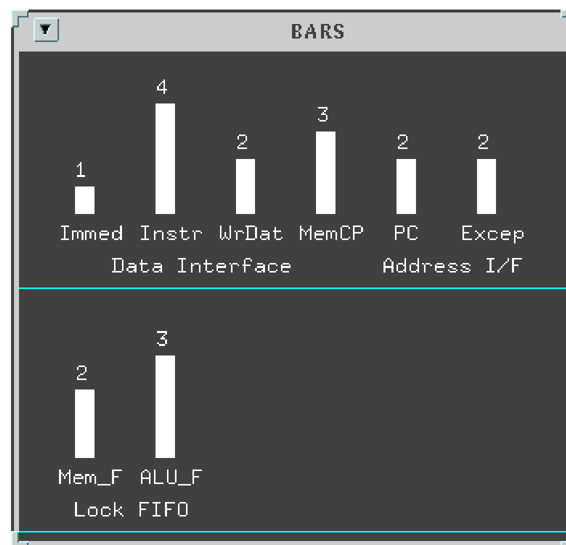


Figure 42 : Pipeline Occupancy using Verilog gr_bars() system task.

The *gr_bars* facility can be used in conjunction with the Pipeline Occupancy module (see Section 8.5.1) to display information regarding the occupancy of all buffering structures used throughout the Asynchronous ARM design. Figure 42 shows the occupancy of the pipelines and buffer structures used in the processor core at a particular instant in (simulated) time. These include: the Immediate Field Extraction Unit, Instruction Pipe, Write Data Buffer and Memory Control Pipe in the Data Interface; the PC Pipe and Exception Pipe in the Address Interface and the Memory Lock FIFO and ALU Lock FIFO in the Register Bank.

8.5.3 Detecting Incorrect Operation

One of the primary requirements in exercising a simulation model of a prototype system is to determine when, and where, the system functions incorrectly. For an asynchronous design, erroneous operation may be easier to detect and locate than for a synchronous design, since, in many cases, the asynchronous system will deadlock. A control element may have generated a request signal and not received an acknowledge due to a design fault in the control circuit. Location of the fault is usually achieved by determining which request-acknowledge signalling pairs have not yet completed their communication actions and examining the control circuits responsible for generating these signals.

In some circumstances, the control signal events going to a particular control element may not appear in the correct sequence. For example, an Arbiter may receive a request signal on input R1 and then receive a second request event on R1 before a done signal is received (D1), releasing the Arbiter after the first R1 request. Also, for a Call element, the common (subroutine) done signal may be received before any of the request input channels has actually received a request event. Generally, the cause of incorrect sequencing of the control signals is (as above) design faults in the control circuits. The Verilog behavioural models of many of the asynchronous control elements contain extra checks to detect incorrect interface signal sequencing and report errors (including the module instance concerned and the time) while the simulation is running. Also, when incorrect sequencing is detected, the outputs of the particular control element are forced into the undefined state, since in the real system the output values would not be valid if the control element functions incorrectly.

In an asynchronous system composed of functional units communicating using transition signalling, an event occurs when the logic value on any signal wire changes between the logic 0 and logic 1 levels - in either direction. An undefined value on any of the control signal wires in such a system could prove catastrophic, particularly if the undefined state remains undetected. Usually, an undefined control signal causes the request-acknowledge communication protocol to fail and the system will deadlock. In or-

der to ensure that the system deadlocks as quickly as possible in such circumstances, the Verilog models of the control blocks of the MDCML Asynchronous ARM have been written so that undefined control signals are rapidly propagated throughout the system. Any undefined input signal arriving at a control element is immediately propagated to all outputs of that control element. Total system deadlock results very quickly, enabling the source of the original undefined signal to be easily detected.

The case of incorrect operation which is most difficult to detect is where the bundling constraint is not met when a data value is passed between two asynchronous modules using the bundled-data interface. If the transfer request event arrives at the receiving module before the actual data value, the receiver may latch (capture) an incorrect data value. The request and acknowledge control signals are correctly generated and received by the sender and receiver, respectively, and in the correct sequence. As a result, the system will continue to operate, but with the ‘wrong’ data value. The effects of propagating an incorrect data value may be significant, particularly if the value is subsequently used to generate system control signals. It is in consideration of this factor that a great deal of design effort must be directed towards eliminating ‘data bundling’ errors. The Bundle Checker module (section 8.5.1) assists the asynchronous logic designer appreciably.

8.6 Performance analysis

8.6.1 Subsystem Processing Performance

The Dhrystone benchmark program has been used as a general test program to evaluate alternative design decisions and to provide a performance measure. In particular, it allows the effect of a change in processing rate of a given datapath component, in the context of overall system performance, to be assessed in order to pinpoint computational bottlenecks. The effect on the execution time of varying a module’s processing rate by

altering the delay between its Request_In and Request_Out control signals is shown in Figure 43 for the ALU, Register Bank and Primary Decode PLA.

It might be expected that at a high processing rate (for a given subsystem block) the graphs would be almost horizontal until a point was reached, as the processing rate decreased, when the time delay through the block would move it onto the 'critical path' causing system performance to be severely impacted.

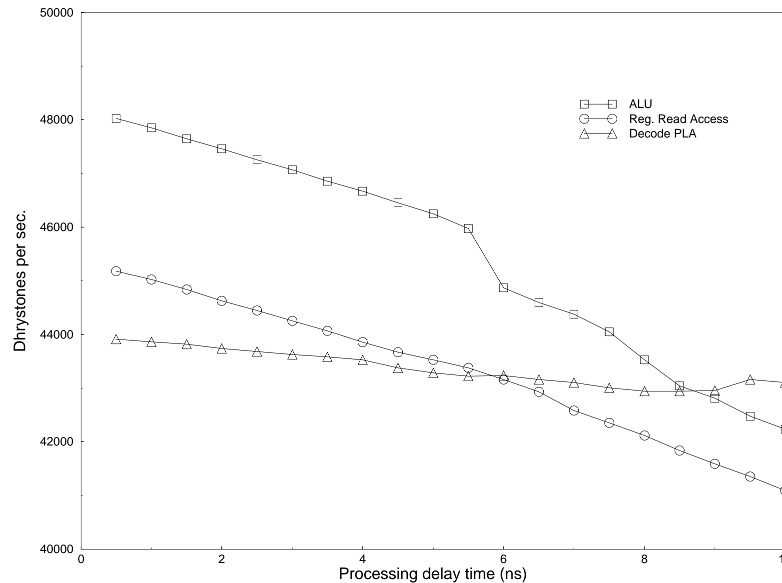


Figure 43 : Graph of Block Processing Time vs Dhrystone performance.

The results however do not show this. Instead they seem to suggest that in an asynchronous system of inter-communicating modules, when considered over a number of executed instructions, every subsystem on the datapath is on the 'critical path', i.e. a change in processing rate of any subsystem has an effect on overall system performance. The Dhrystone performance graph for the Primary Decode PLA is approximately constant over the processing delay range shown. This tends to indicate almost complete overlap with concurrent, slower datapath operation. By considering the gradient of the graphs for each subsystem, it can be noted that the degree of linkage between subsystem performance and overall system performance is different. Design effort, to increase system performance, should therefore be concentrated on those subsystems which produce the

steepest gradient processing rate graphs, since this will have the most beneficial effect on overall instruction throughput.

8.6.2 Non-symmetrical Propagation Delays

Due to the characteristics of the underlying bipolar technology, with inputs to the basic circuit elements being at different voltage levels, the propagation delay from input to output for many of the primitive logic functions is non-symmetrical i.e. it is different for each of the inputs. To determine if the effect of the non-symmetrical propagation delays significantly affects the performance of the MDCML Asynchronous ARM, the switching characteristics of two of the most heavily used primitive asynchronous control elements, the XOR gate and the Muller-C element, are examined.

The XOR gate acts as a MERGE element for events (see Section 3.1.1). An output event (transition) is generated for **every** input event. Initially, the most active input of each XOR gate instance is determined i.e. the input that switches most during a complete run of the benchmark program. The most active input signal is then assigned to the fastest switching (level 3) input terminal of each of the XOR gates and the benchmark is again run. For comparison, the XOR gate inputs are reversed (with the most active input signal assigned to the slower switching input terminal, at level 2) and the benchmark program is again executed.

The Muller-C gate acts as a JOIN element for events (see Section 3.1.2). An output event is generated only after an event has been received on **both** inputs. In contrast to the ‘most active input’ technique for the XOR gate, the ‘later switching’ input must be determined for each Muller-C element i.e. the last event to arrive for each input event ‘pair’. The later switching input signal is assigned to the fastest switching (again level 3) input terminal of the Muller-C and the benchmark program executed. Again, for comparison, the benchmark is executed with the Muller-C elements inputs reversed.

The results for the XOR gate and Muller-C element are shown in the table below (all figures are expressed in Dhrystones per second):

	Fastest	Slowest	Difference
XOR	43535	43137	398
MULLER-C	43950	42644	1306

Figure 44 : Effect of Non-Symmetrical Propagation Delays.

The figures indicate that, in the case of the XOR gate, the non-symmetrical propagation delays have little effect on overall performance. However, in the case of the Muller-C element, a 3% improvement in system performance can be achieved simply by connecting the gate the "optimum way round".

8.6.3 Processor-Memory Interaction

The MDCML Asynchronous ARM processor core is contained within an external simulation environment which includes a simple MMU and memory model capable of supporting the bundled-data communication protocol. In order to determine if the processor performance is limited by the external memory access time, for prefetching instructions or reading and writing data values, several simulation runs of the Dhrystone benchmark program were performed with different access time values in the memory model for each run. The results are shown in the Figure 45 overleaf.

The graph shows that the processor performance is, to some extent, limited by the external memory speed. Although a doubling of memory speed does not result in a doubling of processor speed, any increased memory performance is reflected in increased processor performance. Also, an indication of the peak performance of the processor can be obtained by extrapolating the graph backwards to the zero point on the x-axis, i.e. memory access time is 0ns (an infinitely fast memory). This gives a theoretical peak performance of around 46,500 Dhrystones per second.

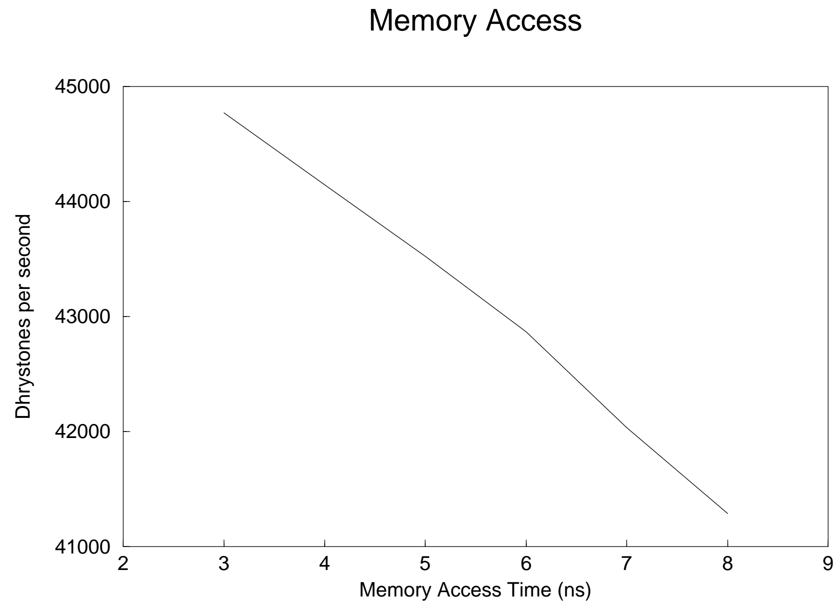


Figure 45 : Effect of Memory Speed on Processor Performance.

8.6.4 Internal Pipeline Efficiency

The Asynchronous ARM processor contains several pipeline structures which act as buffers to even out the flow of data between functional units of differing speed in the design. Some Event Registers between datapath stages are necessary to support concurrent operation since a previous result can be held while a unit computes its next result. In an attempt to improve the performance of the overall system, the efficiency of these pipelines must be examined. The lengths of some of the internal processor pipelines are fixed, since they perform a particular function or are used to prevent potential deadlock situations. For example, the PC Pipe in the Address Interface must be 2 stages long (see Section 7.2.4) and the 5-stage Instruction FIFO Pipeline in the Data Interface must be 3 stages longer than the PC Pipe to prevent a complex deadlock state (see Section 7.2.5). Also, the Memory Control Pipe in the Data Interface must be the same length as the Instruction FIFO Pipeline.

The operation of 4 particular pipelines will be examined in detail. These are the ALU and Memory Lock FIFOs in the Register Bank, the Immediate Field Extraction Unit and the Write Data buffering structure in the DOUT section of the Data Interface.

The Pipeline Occupancy Monitor module was connected to the external request and acknowledge signals of the pipelines under investigation and the Dhrystone benchmark program was executed. The results are displayed, using the *$gr_bars()* system task, in Figure 46 below:

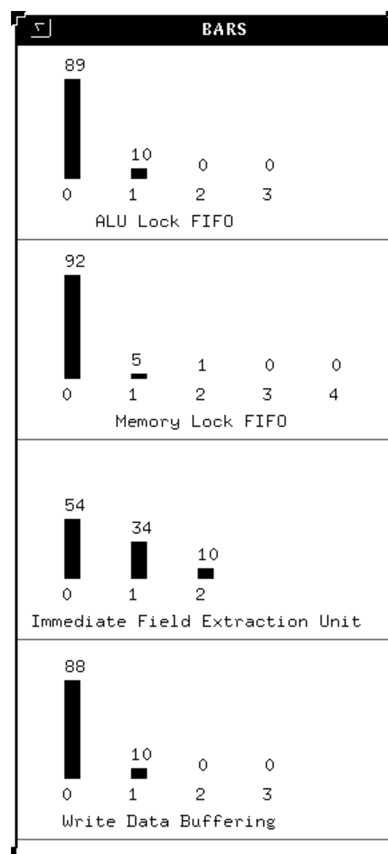


Figure 46 : Pipeline Occupancy during Benchmark Execution.

For each of the pipelines, the fraction of the total simulation time that the pipeline occupancy was a particular value is shown. For example, for 89% of the total time, the ALU Lock FIFO was empty and for 10% of the time, the ALU Lock FIFO contained only one item.

The results seem to suggest that the ALU Lock FIFO, Memory Lock FIFO and Write Data Buffering pipelines are too long and could be reduced to contain only 1 stage (or

possibly removed altogether). The Immediate Field Extraction Unit is probably the correct length.

The investigation was carried further by modifying the length of each of the pipelines, in isolation, and noting the effect on the processor performance when executing the Dhrystone benchmark. Performance may be improved by shortening pipelines, which reduces the latency of the pipeline, i.e. the time taken for a single item to pass through an empty pipeline. The results are shown in the graphs below in Figure 47.

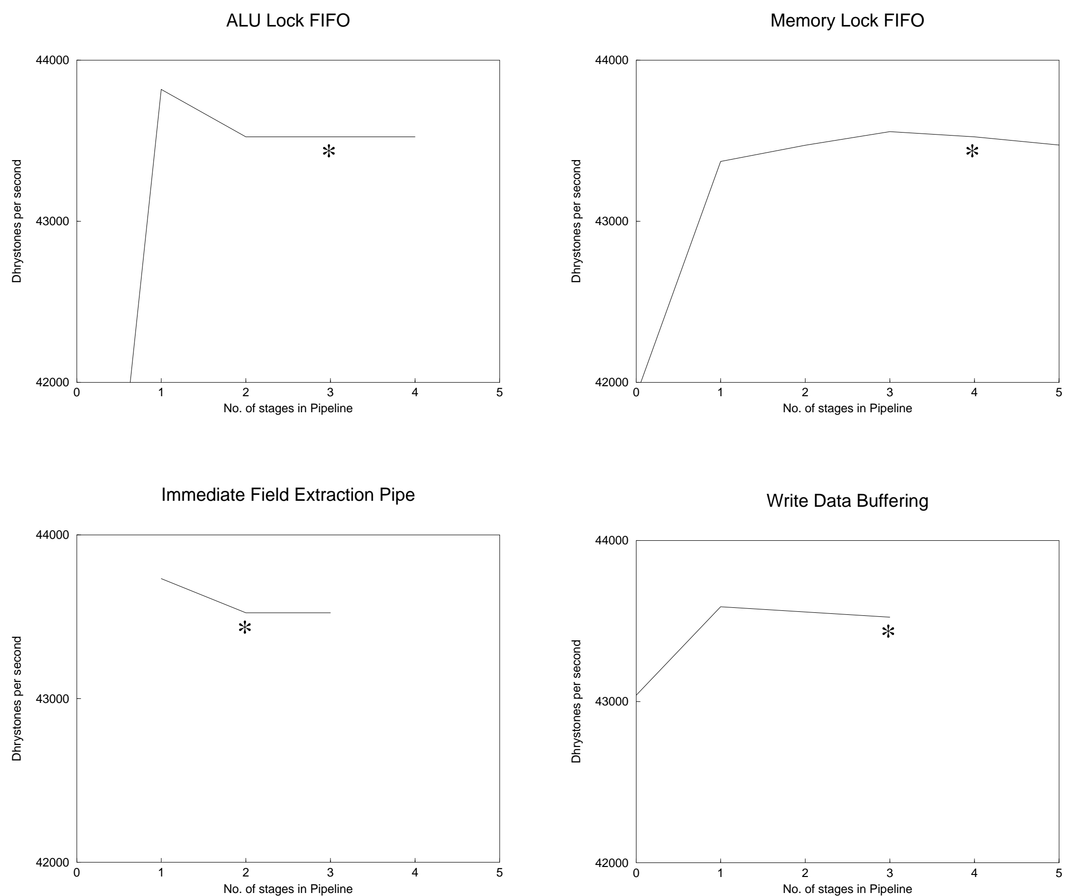


Figure 47 : Effect of Pipeline Length on Processor Performance.

The *'s in each of the graphs shows the length of that particular structure in the current MDCML Asynchronous ARM design. Note that the Immediate Field Extraction Pipe must contain at least one stage for correct system operation.

The results indicate that the ALU Lock FIFO should be shortened by 2 stages (to 1 stage), the Memory Lock FIFO should be shortened by 1 stage (to 3 stages), the Immediate Field Extraction Pipe should be shortened by 1 stage (to 1 stage) and the Write Data Buffer should contain only 1 stage (shortened by 2 stages). These 'recommended' pipeline modifications were simultaneously incorporated into the processor design and the benchmark was again executed. The resulting Dhrystone performance was measured at 44,045 Dhrystones per second - the increase in performance was approximately equal to the sum of the performance increases when the best case of each individual pipeline graph is considered separately. Of greater importance in the bipolar design, than the performance gain, is the decrease in silicon area used for active cells and routing in addition to the power reduction.

The conclusion from this pipeline efficiency study is that the lengths of the pipelines in the current MDCML Asynchronous ARM design should be reduced, in some cases by as much as 2 stages. However, the results only apply to the execution performance of a particular program (the Dhrystone benchmark). There is a requirement to consider a range of general-purpose applications, where individual pipeline structures may be more heavily stressed and, unless silicon area is at a premium, it is better to provide extra buffering to smooth out processing 'hot spots'.

8.6.5 Comments on the Performance Analysis.

As mentioned in the concluding comments of the previous chapter, developing an understanding of the total system operation of the MDCML Asynchronous ARM, with its complex integration of inter-communicating, self-timed subsystems is still in its early stages. However, the ability to develop user-instrumentation for a wide variety of monitoring tasks using the Verilog behavioural modelling language and to present the resulting information in its most appropriate form using the graphical and text output Verilog system tasks assists the asynchronous logic designer appreciably in designing working (i.e. correct) systems and exploring the dynamic behaviour of those systems.

9. Conclusions

The principal aim of this project was to build an architectural model of the MDCML (bipolar) Asynchronous ARM processor capable of supporting the simulated execution of real ARM instruction code programs. This has been achieved. Furthermore, the model has then been used to explore the dynamic behaviour of the system. Various forms of user-instrumentation were written by the author to enable detailed examination of particular function units, and to present the resulting information in a wide variety of forms. Design enhancements were then proposed and tested by the execution of a widely-used benchmark program.

When designing systems incorporating new ideas, whether these are implementation technology developments or new architectural features, the risks of encountering difficulties are increased over a more mature foundry process or circuit design style. Simulation offers the opportunity to exhaustively test the prototype system before it is committed to the integrated circuit manufacture, where design changes are not possible.

9.1 Production of the System Model

Initially, circuit simulation of the basic bipolar logic primitives was carried out to provide information regarding the switching characteristics of the target implementation technology. The knowledge gained was then employed to construct structural and behavioural models of the standard logic primitives (AND, OR, etc.) and asynchronous control elements in the Verilog modelling environment. By producing gate-level and

functional models of the system building blocks, the simulation of a large-scale processor design becomes computationally feasible.

The functional subsystems of the Asynchronous ARM, including the Register Bank, ALU, Memory Interface and Decode logic were then developed. They were constructed, either from the combination of the logic primitives and asynchronous control elements or from representations involving a behavioural description. The complete system consists of the functional subsystems supported by a transition-signalling communication protocol.

9.2 Current State of the Project

The architectural model of the MDCML Asynchronous ARM processor has been completed and, at the time of writing, the major part of the datapath is near submission for fabrication. This has only been possible through the use of simulation since it involves a novel design methodology and a new target implementation technology. A simulation environment, consisting of a simple Memory Management Unit (MMU) and an external memory model, has also been produced. The Asynchronous ARM model successfully executes all the programs in the ARM Validation Suite, except for those instructions requiring specific hardware resources which will not be implemented in the target bipolar technology. A number of design and monitoring aids have been written by the author which expose significant parts of the internal operation of the asynchronous system. Information gained during the processor pipeline length investigation enabled the length of the ALU and Memory Lock FIFOs to be reduced in the original design to improve performance and reduce silicon area.

9.3 Comments on the Verilog Modelling Environment

Verilog provides an ideal environment for modelling a micropipelined asynchronous microprocessor architecture. Its modular, hierarchical structure is in harmony with a system composed of inter-communicating asynchronous functional units, and asyn-

chronous operation maps well onto its control constructs. The flexibility and suitability of Verilog is further demonstrated by the production of custom tools and test vectors specifically for our prototype design.

The bottom-up, incremental design style and verification of individual primitive components is easily accommodated into a high-level, behavioural view of the overall system, which is largely technology independent. The production of a full execution code-compatible architectural model results in a valuable aid in analysing the dynamic behaviour of the system and gives a degree of confidence in the design approach. Alternative design decisions have been more easily evaluated and an indication of expected performance has been gained.

In common with many other digital logic modelling environments, a Verilog design description is exercised by means of an event-driven simulator. This simulation paradigm fits particularly well with the event-driven computational model of asynchronous logic. Furthermore, the timing control mechanisms incorporated into the Verilog behavioural language, especially the **event control** constructs, would be ideal for modelling a self-timed system developed using any asynchronous design methodology.

A high degree of concurrency is supported in the Verilog system model through the use of the **fork** and **join** compound statements (see Section 4.5.1), allowing a non-deterministic ordering of the notionally parallel execution of the individual statements. Also, multiple **always** @ event control blocks across the entire design result in many 'threads of execution' being simultaneously active throughout the prototype system.

The modular approach to designing with asynchronous inter-communicating subsystems afforded by the micropipeline approach is closely reflected in the architectural modelling environment of Verilog with its hierarchical module structure. All these features make Verilog sympathetic to an asynchronous design style.

9.4 Future Research

9.4.1 Technology Migration

The architectural modelling of the MDCML Asynchronous ARM design has been achieved in a hierarchical, modular fashion at a relatively high level of abstraction. The figures used for the propagation delays of the standard logic primitives and asynchronous control elements are based on the circuit simulation of their realisation in the target bipolar technology. By re-designing the required low-level components in a different implementation technology and determining the respective propagation delay times, the characteristics of the new target technology can be incorporated into the basic Verilog system models. This results in the Asynchronous ARM processor design being easily migrated to a new fabrication technology and would, for example, enable a comparison between MDCML and CMOS on the basis of performance.

Of course, detailed design of the functional units will consider if any circuit optimisations exist in the new implementation technology to increase the performance, reduce the gate count, or improve the power efficiency of the system. For example, the lack of a Wire-OR circuit design technique in the MDCML bipolar technology significantly increased the amount of logic required and the propagation delay times for the ALU Completion logic and the Zero-Detect function in the current ALU design. Although the basic switching speed of the MDCML technology is superior to that of CMOS, the circuit design flexibility afforded by CMOS can produce faster and smaller component designs in certain circumstances.

9.4.2 Architectural Design Alternatives

In producing the MDCML Asynchronous ARM design, various datapath functional units and control circuit components have been developed using the behavioural modelling language of the Verilog environment. The vast amount of simulation and valida-

tion performed on the system containing these components should convince the logic designer of the integrity of these components.

The system designer is now free to compose these datapath and control elements to produce and explore novel asynchronous computational structures. Multiple functional units can be combined to produce an asynchronous superscalar design or more radical architectures, such as dataflow, may be considered.

References

[ARM91] *ARM6 Macrocell datasheet.* ARM Ltd., Cambridge, England, September, 1991.

[Brow91] Brown A., *VLSI Circuits and Systems in Silicon.* McGraw-Hill, ISBN 0-07-707221-9, 1991.

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

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

[Depe89] Depey M.P. et al., A 10K-Gate 950-MHz CML Demonstrator Circuit Made with a 1-mm Trench-Isolated Bipolar Silicon Technology. *IEEE Journal of Solid-State Circuits,* 24(3):552-557, June, 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.

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

[Furb94a] Furber S.B., Day P., Garside J.D., Paver N.C., Woods J.V., AMULET1: A Micropipelined ARM. *Proceedings of IEEE Computer Conference (CompCon'94),* San Francisco, USA, March, 1994.

[Furb94b] Furber S.B., Day P., Garside J.D., Paver N.C., Temple S., Woods J.V., The Design and Evaluation of an Asynchronous Microprocessor. *IEEE International Conference on Computer Design (ICCD ‘94),* October, 1994.

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

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

[GPS88] GEC - Plessey Semiconductors, *Differential Logic Design Manual (FAB 4) 1.0 Edition.* July, 1988.

[Hart87] Hartenstein R., *Hardware Description Languages.* Advances in CAD for VLSI Series, Vol. 7, Elsevier Science Publishers B.V., ISBN 0--707221-9, 1987.

[Hauc93] Hauck S., *Asynchronous Design Methodologies: An Overview.* Tech. Rep. 93-05-07, Dept. of Computer Science and Engineering, University of Washington, U.S.A. 1993.

[Henn90] Hennessy J.L., Patterson D.A., *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Mateo. CA. 1990.

[Hill87] Hill D.D., Coelho D.R., *Multi-level Simulation for VLSI Design.* Kluwer Academic Publishers, ISBN 0-89838-184-3, 1987.

[Hoar85] Hoare C.A.R., *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Hspi90] *HSPICE User Manual,* Meta-Software Inc., CA.

[Jagg90] Jagger D.V., *A Performance Study of the Acorn RISC Machine.* M.Sc. Thesis, University of Canterbury, New Zealand, 1990.

[John91] Johnson M., *Superscalar Microprocessor Design.* Prentice-Hall International, ISBN 0-13-875634-1, 1991.

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

[Kogg82] Kogge P.M., *The Architecture of Pipelined Computers.* Hemisphere, 1982.

[Mano84] Mano M.M., *Digital Design.* Prentice-Hall International, ISBN 0-13-212325-8, 1984.

[Mart89] 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.

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

[Moln92] Molnar C.E., Jones I., Sutherland I.E., *A Way to Compose Petri Nets.* Tech. Rep. SMLI #92:0354, Sun Microsystems Inc., October, 1992.

[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.

[Pave91] Paver N.C., *Condition Detection in Asynchronous Pipelines.* UK Patent no 9114513, October, 1991.

[Pave92] Paver N.C., Day P., Furber S.B., Garside J.D., Woods J.V., Register Locking in an Asynchronous Microprocessor. *Proceedings of ICCD '92,* pp 351-355, October, 1992.

[Pave94] Paver N.C., *The Design and Implementation of an Asynchronous Microprocessor.* PhD thesis, University of Manchester, June, 1994.

[Pete81] Peterson J., *Petri net theory and modelling of systems.* Prentice-Hall, 1981.

[Russ85] Russell G., Kinniment D.J., Chester E.G., McLauchlan M.R., *CAD for VLSI.* Van Nostrand Reinhold (UK), ISBN 0-442-30618-0, 1985.

[Russ89] Russell G., Sayers I.L., *Advanced Simulation and Test Methodologies for VLSI Design.* Van Nostrand Reinhold (Int.), ISBN 0-7476-0001-5, 1989.

[Seit80] Seitz C.L., System Timing. In *Introduction to VLSI Systems,* editors Mead C.A., Conway L.A., Chapter 7, Addison-Wesley, 1980.

[Suth86] Sutherland I.E., Sproull R.F., *Asynchronous Systems.* Sutherland, Sproull & Associates, Palo Alto, California, September, 1986.

[Suth89] Sutherland I.E., Micropipelines. *Communications of the ACM.* 32(6):720-738, January, 1989.

[Thom92] Thomas D.E., Moorby P., *The Verilog Hardware Description Language.* Kluwer Academic Publishers, ISBN 0-7923-9126-8, 1992.

[Veri92] *Verilog-XL Reference Manual, Volumes 1&2.* Cadence Design Systems Inc., 1992

[VLSI90] *Acorn Risc Machine (ARM) Family Data Manual.* VLSI Technology Inc., Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[Weic84] Weicker R.P., Dhrystone: A Synthetic Systems Programming Benchmark. *Communications of the ACM.* 27(10):1013-1030, October, 1984.

[West89] Weste N., Eshraghian K., *Principles of CMOS Design - A Systems Perspective.* Addison-Wesley, Wokingham, England, 1989.

Appendix A: Verilog Model

The following complete listing of the MDCML (bipolar) Asynchronous ARM Verilog Model contains:

Top-level processor core 143

ARM functions behavioural library 152

Asynchronous component library 159

Standard gate functions library 168

Example of PLA structure modelling 173

External environment model - MMU and memory 174

```
//*************************************************************************************//
//                                                                                     //
//                 The following file contains the complete hierarchical              //
//                      model of the Asynchronous ARM processor core.                 //
//                                                                                     //
//*************************************************************************************//

//      Micropipelined ARM
`timescale 10ps /10ps

module ARMstCore (Add[31:0],Dout[31:0],Ctl[6:0],MEMr,MRRa,
                              MRR[31:0],Nfiq,Nirq,Dabt[1:0],PAbt,MEMa,MRRr,BigEnd,nAbt,Rst);
`include "ARMstCore.inc"

//      Ctl[] = Seq, Inc, Ren, Wen, Usr, B/W, Opc

// memory data in blocks and instruction pipeline
DatInt dat (MARa,DOa,MEMr2,MWR[31:0],MRRa,DWr,DW[31:0],DWusr,
            DWv,DWpc,INr,IN[31:0],flo[1:0],Nim[31:0],IMa0,IMr2,
                                 MARr,DOr,nMLb[31:0],DObw,MEMa2,MRRr,MRR[31:0],PAbt,DWa,
                                INa,{MAR[1:0],MAc[4:0],Mval,MdPC,MpPC},IMr0,IMa2,BigEnd,Rst);

EvtReg #32 lat0 (Dout[31:0],MEMa0,MEMr0,    MWR[31:0],MEMr2,MEMa,Rst);

Cgate2 c0 (MEMa2,    MEMa0,MEMa1,Rst);
Cgate2 c1 (MEMr,     MEMr0,MEMr1,Rst);

// 1st decode stage
Decode1 dec1 (RGr,Rs[27:0],IN2r,IN2[25:0],IN3r,IN3[19:0],PCsel,XPr,
              XLa,INa,IMr0,LSMPr,nTRM,r15,NGr0,nGn[1:0],vect[2:0],
                                    INr,IN[31:0],flo[1:0],Nfiq,Nirq,PCpar,RGa,IN2a,IN3a,XPa,
                                   XLr,PPr3,IMa0,LSMPa,ALUgo,ALUok,mode[5:0],NGa0,nAbt,Rst);

// 1st execution and 2nd decode stage
Reg rg (RGa,RWa,RDr,Na[31:0],Nb[31:0],
                                 RGr,Rs[27:0],nPC[31:2],RWr,W[31:0],Wc[2],Wsel,RDa,Rst);

NGen nGen (NGa0,NGr2,ng[5:0],    NGr0,IN[15:0],nGn[1:0],vect[2:0],NGa2,Rst);

Decode2 dec2 (IN2a,RSa,C2r,{Imd[6:0],SHop[9:0],DObw,c2[7:0]},
                                                     IN2r,IN2[25:0],RSr,Na[7:0],C2a,Rst);

//      IN[]  = Xt[1:0],PCpar,cond[3:0],sctls[2:0],I[11:5],DObw,
//                                                     toRs,cpCP,~toDO,~toA,nGen,~Mult,NImm
//      Imd[] = Xt[1:0],PCpar,cond[3:0]
//      c2[]  = toRs,cpCP,~toDO,~toA,nGen,~Mult,NImm

// 3rd decode stage
Decode3 dec3 (IN3a,C3r,ALfs[9:0],{vec3[2:0],c3[22:0]},    IN3r,IN3[19:0],C3a,Rst);

//     c3[] = UseCP,S,F,C,Wcp[2:0],Ral,Rcnd,~ALUwt,~DabtWt,tPCp[1:0],
//                                              Wreg,Wadd,SP,LSM,Ren,Wen,B/W,Opc,destPC,Rsel

// 3rd control and execution stages
Shift shft (Sh[31:0],ShC,SHd,    nMLb[31:0],Nim[31:0],c2[0],SHop[9:0],psrC,SHe);

ExecP excP (RDa,C2a,C3a,NGa2,SHe,IMa2,PCpar,ALUgo,ALUok,mode[5:0],psrC,
            WRr,W[31:0],Wc[9:0],Wq[1:0],APr,DOr,nMLb[31:0],RSr,Dabt0,
                                   RDr,Na[31:0],Nb[31:0],C2r,c2[7:0],C3r,c3[22:0],vec3[2:0],
                                   ALfs[9:0],NGr2,ng[5:0],Sh[31:0],ShC,SHd,IMr2,Imd[6:0],
                                   DW[31:0],DWv,DWusr,WRa,Wsel,APa,DOa,RSa,Dabt[1:0],Rst);

// write bus control
WrCtl wctl (DWa,WRa,RWr,ADr,Wsel,    DWr,DWpc,DWv,WRr,Wq[1:0],RWa,ADa,WLx,Rst);

// the memory address interface
AddInt add (ADa,WLx,APa,PPr3,XPa,XLr,nPC[31:2],LSMPa,
            MARr,MAR[31:0],{MAc[6:0],Mval,MdPC,MpPC},
                                          ADr,W[31:0],Wc[9:0],APr,Na[31:0],LSMPr,nTRM,
                                          r15,INa,XPr,XLa,PCsel,MARa,Dabt[1],Dabt0,Rst);

//      MAc[] = Seq, Inc, Ren, Wen, Usr, B/W, Opc
//      Mval = valid    MdPC = destPC    MpPC = PCpar

EvtReg2 #(32,7) lat1 (Add[31:0],Ctl[6:0],MEMa1,MEMr1,    MAR[31:0],MAc[6:0],MEMr2,MEMa,Rst);

endmodule // ARMstCore


// Primary Instruction decode logic
module Decode1 (RGr,Rs[27:0],I2r,I2[25:0],I3r,I3[19:0],Dabt,XPr,
                XLa,Ia,IMr,LSMPr,nTRM,r15,NGr,nGn[1:0],vect[2:0],
                                      Ir,I0[31:0],flo[1:0],Nfiq,Nirq,PCpr,RGa,I2a,I3a,XPa,
                                     XLr,PCr,IMa,LSMPa,ALUgo,ALUok,mode[5:0],NGa,nAbt,Rst);
`include "Decode1.inc"

// Instruction Disassembler
//ARM_Dis dis (DIS, I0[31:0], Ir2, Rst);

// arbitrate the exceptions and prioritize (NB mode[4] is omitted!)
DLor2     sg0       (nFiq,    Nfiq,mode[4]);
DLinv     inv0      (Fiq,     nFiq);
DLor2     sg1       (nIiq,    Nirq,mode[5]);
DLinv     inv1      (Iiq,     nIiq);
ArbitX    arb       (Frq,Irq,PAR,Ir0,    Fiq,Iiq,PCpr,Ia,Ia,Rst);
XcpC_PLA #100 dec0  (I[27:24],I[4],Frq,Irq,Dabt,flo[1],    nXR,vect[2:0],{Xt1,Xt0});

// check PC parity, and reject if wrong
DLxor2    x0        (NpOK,    PAR,flo[0]);
Select    sl0       (Ir2,Irj, Ir1,NpOK,Rst);
DLxor2    x1        (Ia,      Irj,Ia4);

// delay Data Abort request until ready
EvtLch    el0       (Ipar,    flo[0],Ir,Rst);
DLxor2    x00       (NpOK1,   PAR,Ipar);
DLor2     sg3       (NpOK2,   NpOK1,Rst);
LatchR    lt0       (DABr,    XLr,NpOK2,Rst);
DLxor2    x2        (Dabt,    DABr,XLa);
Select    sl1       (Ia4,XLa, Ia3,Dabt,Rst);
DLxor2    x3        (Ir3,     DABr,Ir2);

// force I[] into a known state during Dabts
DLinv     sg4       (Z,       Dabt);
Band #32  Iz        (I[31:0], I0[31:0],{32{Z}});

// sync with the PC and exceptions
Cgate3    c0        (Ir1,     Ir0,Ir,PCr,Rst);
```

```
// the first stage decoders
// ip[] = ALct[3:0], sctls[2:0], BWen, ~Rs, cpCP, ~toDO, ~toA, ~Mult
RegC_DPLA #400 dec1 ({nXR,Rs[4],I[27:20],I[7],I[4],sq0[1:0]},Ir4,Rst,
                               {P[7:0],Rs[27:26],Rs[15],NImm,NRdG,NtoX0,nGn[1:0],
                               toN,nALw1,nALw0,sq1[2:0],toI3,ip[12:0]},Pr);
DLor2     sgn       (NtoX,         nAbt,NtoX0);

// the multicycle sequencer
seqC_PLA #200 seq (sq1[2:0],ALUok,I[24],I[21:20],P[7],r15A,Rs[4],sq2[1:0],multi);
EvtLch #2 el1       (sq0[1:0],     sq2[1:0],Ia2,Rst);
Select    sl2       (Ia3,Irm,      Ia2,multi,Rst);
DLxor2    x5        (Ir4,Ir3,      Irm);

// the register address muxes
mux4 #4   mx0       (Rs[25:22],    pc[3:0],I[19:16],I[11:8],pc[3:0], P[1:0]);
mux4 #4   mx1       (Rs[21:18],    Rd[3:0],I[19:16],I[15:12],I[3:0], P[3:2]);
mux4 #4   mx2       (Rs[14:11],    pc[3:0],Rd[3:0],I[15:12],pc[3:0], P[5:4]);
mux4 #4   mx3       (Rs[10:7],     pc[3:0],lk[3:0],I[19:16],I[15:12], P[7:6]);

// detect PC targets
DLand4    g00       (r15A,         Rs[10],Rs[9],Rs[8],Rs[7]);
DLand4    g01       (r15M,         Rs[14],Rs[13],Rs[12],Rs[11]);

// the bank select logic
NbsC_PLA #100 dec2  (mode[3:0],NRdG,I[22],I[20],I[15],      Rs[6:0]);

// load & store multiple
Select    sl3       (RdGr,Pr1,     Pr,NRdG,Rst);
RdGen     RdGn      (RdGa,LSMPr0,Rd[3:0],nTRM0,r150,   RdGr,RdGa1,I[15:0],Rst);
// NB Rd[] = 15 when RdGen inactive !
Cgate3    c1        (RdGa1,        RGa1,I2a2,LSMPa0,Rst);
Select    sl4       (I2r1,noI2,    LSMPr0,I[20],Rst);       // I[20] = LSM ld/not store
DLxor2    x6        (I2a2,         I2a1,noI2);
DLxor2    x7        (Ia2,          Ia1,RdGa);

// the LSM pipe
EvtReg #2 ls0       ({nTRM,r15},LSMPa0,LSMPr,     {nTRM0,r150},LSMPr0,LSMPa,Rst);

// propagate the instruction to decode 2
// I2[] = Xt[1:0],PCpar,cond[3:0],sctls[2:0],I[11:5],
//                                             DObw,toI3,~Rs,cpCP,~toDO,toA,toN,~Mult, NImm
Call      ca0       (I2x,I2a1,I2r0,Pr1,I2r1,I2a0,Rst);
DLand2    g1        (BW,           ip[5],I[22]);
EvtReg #26 l2       (I2[25:0],I2a0,I2r,       {Xt1,Xt0,Ipar,I[31:28],ip[8:6],I[11:5],BW,
                                             toI3,ip[4:1],toN,ip[0],NImm},I2r0,I2a,Rst);

// propagate the instruction to decode 3
// I3[] = vect[2:0], r15M, r15A, I[24:19,16:15], ALct[3:0], ~ALUwt, ~DabtWt, B/W
Select    sl5       (noI3,I3r0,    Pr1,toI3,Rst);
DLxor2    x8        (I3x,          noI3,I3a0);
EvtReg #20l3        (I3[19:0],I3a0,I3r,       {vect[2:0],r15M,r15A,I[24:19],I[16:15],
                                             ip[12:9],nALw0,NtoX,BW},I3r0,I3a,Rst);

// request the register operation
// Rs[] = RdEn, ArNp, Ra[3:0], Rb[3:0], Psel[1:0], LrNp, Rm[3:0], Rw[3:0], NBS[6:0]
PslC_PLA #100 dec3  (nXR,I[19],I[16],         Rs[17:16]);
Call      ca1       (RGx,RGa1,RGr,            Pr1,LSMPr0,RGa,Rst);
```

```
// kick off the immediate pipeline if needed
Select    sl6       (IMr,noImm,    Pr1,NImm,Rst);
DLxor2    x9        (IMx,          IMa,noImm);

// copy PC to X pipe if a load or store
Select    sl7       (XPr,noX,      Pr1,NtoX,Rst);
DLxor2    x10       (XPx,          XPa,noX);

// kick off nGen when needed
Select    sl8       (noN,NGr,      Pr1,toN,Rst);
DLxor2    x11       (NGx,          NGa,noN);

// wait for ALU to complete mode change & interrupt disable?
Select    sl9       (ALr,noA,      Pr1,nALw1,Rst);
Cgate2    c2        (ALg,          ALr,ALUgo,Rst);
DLxor2    x12       (ALx,          ALg,noA);

// wait for all outputs to complete
Cgate2    c3        (xpal,         ALx,XPx,Rst);
Cgate2    c4        (imng,         NGx,IMx,Rst);
Cgate2    c5        (I32x,         I2x,I3x,Rst);
Cgate2    c6        (Ia0,          xpal,imng,Rst);
Cgate2    c7        (rg32,         I32x,RGx,Rst);
Cgate2    c8        (Ia1,          Ia0,rg32,Rst);

endmodule // Dec1


module ArbitX (F,I,P,Ir1,          Nfiq,Nirq,PCpr,Ir0,Ia,Rst);
input Nfiq,Nirq,PCpr,Ir0,Ia,Rst;
output F,I,P,Ir1;

// synchronise exception requests with the instruction stream
DLinv     invR      (NRst,         Rst);
DLxor2    x0        (Ir,           Ir0,NRst);
Arbit     a0        (F,rI0,        Nfiq,Ir,F,Ia,Rst);
Arbit     a1        (I,rI1,        Nirq,Ir,I,Ia,Rst);
Arbit     a2        (P,rI2,        PCpr,Ir,P,Ia,Rst);
Cgate3    c         (Ir1,          rI0,rI1,rI2,Rst);

endmodule // ArbitX


// Secondary (shift) decode logic
module Decode2 (IN2a,RSa,C2r,d2[25:0],IN2r,IN2[25:0],RSr,Rs[7:0],C2a,Rst);

`include "Decode2.inc"

// IN2[] = Xt[1:0],PCpar,cond[3:0],sctls[2:0],I[11:5],
//                           DObw,toI3,~Rs,cpCP,~toDO,~toA,nGen,~Mult,NImm}
// d2[] = Xt[1:0],PCpar,cond[3:0],SHop[9:0],
//                           DObw,toI3,~Rs,cpCP,~toDO,~toA,nGen,~Mult,NImm}

EvtReg #8 Rsl       (Rs2[7:0],RSa,RSr2,      Rs[7:0],RSr,RSa2,Rst);

DLor4     g1        (IMror,                  IN2[15],IN2[14],IN2[13],IN2[12]);

ShfC_PLA #100 shc   (IN2[10:9],Rs2[7:0],     shR[9:0]);
```

```
mux4 #10 mx0        (SHop[9:0],
                              {Vdd,Vdd,Vss,Vss,Vss,Vss,Vss,Vss,IN2[18],Vss},// LL0 or 2
                              {Vdd,Vdd,Vss,IN2[10:9],IN2[15:11]},           // inst
                              {Vdd,Vdd,Vss,IMror,IMror,IN2[15:12],Vss},     // imm rotate
                              shR[9:0],                                     // Rs shift
                                        IN2[17:16]);
DLand2    dl0       (RSsh,              IN2[16],IN2[17]);
Select    sl0       (isnt,is,           IN2r,RSsh,Rst);
Cgate2    c0        (isl,               is,RSr2,Rst);
DLxor2    x0        (IN2a,              RSa2,Aisnt);
Call      call      (RSa2,Aisnt,INr1,   is1,isnt,INa2,Rst);
EvtReg #26 l0       (d2[25:0],INa2,C2r, {IN2[25:19],SHop[9:0],IN2[8:0]},INr1,C2a,Rst);

endmodule // Dec2


// Tertiary (ALU) decode logic
module Decode3 (IN3a,C3r,af3[9:0],d3[25:0],  IN3r,IN3[19:0],C3a,Rst);

`include "Decode3.inc"

// IN3[] = vect[2:0],r15M,r15A,I[24:19,16:15],ALct[3:0],~ALUwt,~DabtWt,B/W
// d3[] = vect[2:0],ALfs[9:0],UseCP,S,F,C,wCP[2:0],Ral,Rcnd,~ALUwt,
//                       ~DabtWt,tPCp[1:0],Wreg,Wadd,SP,LSM,Ren,Wen,B/W,Opc,destPC,Rsel

// ALU decode PLA
ALUc_DPLA #400 dec1 ({IN3[16:10],IN3[7:3]},IN3r,Rst,   d[26:0],Dr);

// pipeline the control outputs
EvtReg2 #(10,26) reg0(af2[9:0],d2[25:0],IN3a,r2,
                                        d[26:17],{IN3[19:17],d[16],IN3[10:8],d[15:11],
                                        IN3[2:1],d[10:3],IN3[0],d[2:0]},Dr,a2,Rst);

EvtReg2 #(10,26) reg1(af3[9:0],d3[25:0],a2,C3r,        af2[9:0],d2[25:0],r2,C3a,Rst);

endmodule // Dec3


//   Micropipelined register bank model
module Reg (Ia0,Wak,Drq,Na[31:0],Nb[31:0],
                                   Ir0,I0[27:0],nPC0[31:2],Wrq,W[31:0],Wval,Wma,Dak,Rst);


`include "Reg.inc"

// Irq, Iak bundle I[24:0] and PC[31:2] (the 32-bit PC+8/12 value). I[] is:
// RdEn, ArNp, ADec[3:0], BDec[3:0], PSRsel[1:0], LrNp, LDec[3:0], WDec[3:0], NBS[6:0]

// latch r15 and instruction
EvtReg2 #(30,28) Ilat(nPC[31:2],I[27:0],Ia0,Ir1,nPC0[31:2],I0[27:0],Ir0,Ia1,Rst);

// begin to process next instruction
DLinv     Cinv1     (nPd1,         Pd1);
DLinv     Cinv2     (nLkx,         Lkx);
Cgate3    goC       (Rgo,          Ir1,nPd1,nLkx,Rst);
Cgate2    IakC      (Ia1,          Rd1,Wlak,Rst);
DLxor2    renX      (Ren,          Rg1,Rdn);
```

```
// the read bypass logic
Select    rdby      (Rby,Rg1,      Rgo,I[27],Rst);
DLxor2    rby1      (Rd1,          Rby,Rdn);
DLxor2    rby2      (Pd1,          Rby,Pdn);

// the lock address latch
EvtReg #18 wlat     (L[17:0],Wlak,Wr1,  I[17:0],Rgo,Ldn,Rst);

// the A and B read address decoders and lock gating
Decode    Adec      (dcA[30:0],dcP[4:0],I[25:22],I[6:0],Ren,I[26]);
Band #31  Agate1    (enA[30:0],         dcA[30:0],{Vdd,Lkd[29:0]});
Band #5   Agate2    (psA[4:0],          dcP[4:0],LkP[4:0]);
Decode    Bdec      (dcB[30:0],xx[4:0], I[21:18],I[6:0],Ren,Vdd);
Band #31  Bgate     (enB[30:0],         dcB[30:0],{Vdd,Lkd[29:0]});
Sink #5   skx       (xx[4:0]);                               // SPSR is not read onto B bus

// the lock initiate and bypass logic
Cgate2    LkC       (Lgo,          Rd1,Wr1,Rst);
DLand4    LKa       (nLkALU,       L[10],L[9],L[8],L[7]);
DLinv     LAinv     (LkALU,        nLkALU);
DLand4    LKm       (nLkMEM,       L[14],L[13],L[12],L[11]);
DLinv     LMinv     (LkMEM,        nLkMEM);
Select    LkbyA     (LbyA,LgoA,    Lgo,LkALU,Rst);
Select    LkbyM     (LbyM,LgoM,    Lgo,LkMEM,Rst);
DLxor2    LbyA1     (LxA1,         LbyA,LkxA);
DLxor2    LbyA2     (LdnA,         LbyA,LaA);
DLxor2    LbyM1     (LxM1,         LbyM,LkxM);
DLxor2    LbyM2     (LdnM,         LbyM,LaM);
Cgate2    LkxC      (Lkx,          LxA1,LxM1,Rst);
Cgate2    LkfC      (Ldn,          LdnA,LdnM,Rst);

// the lock address decoder
DLxor2    LnA       (LenA,         LgoA,LaA);
Decode    LdecA     ({Vss,lkA[29:0]},lPA[4:0],    L[10:7],L[6:0],LenA,L[15]);
Toggle    LtogA     (LrA,LkxA,     LenA,Rst);
DLxor2    LnM       (LenM,         LgoM,LaM);
Decode    LdecM     ({Vss,lkM[29:0]},lPM[4:0],    L[14:11],L[6:0],LenM,L[15]);
Toggle    LtogM     (LrM,LkxM,     LenM,Rst);


// the lock FIFO
LkFIFO    LkF       (LaA,LaM,Wgo,Wa[29:0],WPa[4:0],Psel[1:0],Lkd[29:0],LkP[4:0],
                                                  LrA,LrM,lkA[29:0],lkM[29:0],lPA[4:0],
                                                  lPM[4:0],L[17:16],Wrq,Wma,Wak,Rst);

// the write control logic
DLxor2    Wn        (Wen,          Wg1,Wdn);
WriteEn   wren      (enW[29:0],psWm[4:0],psWf[4:0],    Wa[29:0],WPa[4:0],Psel[1:0],Wen);
WrComp    wcmp      (Wcmp,         Wen,enW[29:0],psWm[4:0],psWf[4:0]);
Toggle    Wtog      (Wdn,Wrx,      Wcmp,Rst);

// the write bypass logic
Select    Wrby      (Wby,Wg1,      Wgo,Wval,Rst);
DLxor2    Wbyx      (Wak,          Wby,Wrx);

// the main and SPSR registers
REG       rg        (Na0[31:0],Nb0[31:0],Adn,Bdn,
                                        W[31:0],nPC[31:2],enA[30:0],psA[4:0],
                                        enB[30:0],enW[29:0],psWm[4:0],psWf[4:0],Ren);
```

```
// the read completion and read data latches
Cgate2    abC       (ABdn,         Adn,Bdn,Rst);
Toggle    Rtog      (Dlrq,Pdn,     ABdn,Rst);

EvtReg2 #(32,32) Dlat (Na[31:0],Nb[31:0],Rdn,Drq,      Na0[31:0],Nb0[31:0],Dlrq,Dak,Rst);

endmodule // Reg


module LkFIFO (LaA,LaM,Wgo,Wa[29:0],WPa[4:0],Psel[1:0],Lkd[29:0],LkP[4:0],
                    LrA,LrM,lkA[29:0],lkM[29:0],lPA[4:0],lPM[4:0],Ps0[1:0],Wrq,Wma,Wdn,Rst);

`include "LkFIFO.inc"

// the internal (ALU) sourced result address lock FIFO
LkReg     stg1      (l1[29:0],P1[4:0],Ps1[1:0],lk1[29:0],Pk1[4:0],LaA,rq2,
                              lkA[29:0],lPA[4:0],Ps0[1:0],lkA[29:0],lPA[4:0],LrA,ak2,Rst);
LkReg     stg2      (l2[29:0],P2[4:0],Ps2[1:0],lk2[29:0],Pk2[4:0],ak2,rq3,
                               l1[29:0],P1[4:0],Ps1[1:0],lk1[29:0],Pk1[4:0],rq2,ak3,Rst);
LkReg     stg3      (l3[29:0],P3[4:0],Ps3[1:0],lk3[29:0],Pk3[4:0],ak3,Arq,
                               l2[29:0],P2[4:0],Ps2[1:0],lk2[29:0],Pk2[4:0],rq3,Aak,Rst);

// the external (memory) sourced result address lock FIFO
LkReg     stg4      (l4[29:0],P4[4:0],Ps4[1:0],lk4[29:0],Pk4[4:0],LaM,rq5,
                               lkM[29:0],lPM[4:0],Ps0[1:0],lk3[29:0],Pk3[4:0],LrM,ak5,Rst);
LkReg     stg5      (l5[29:0],P5[4:0],Ps5[1:0],lk5[29:0],Pk5[4:0],ak5,rq6,
                               l4[29:0],P4[4:0],Ps4[1:0],lk4[29:0],Pk4[4:0],rq5,ak6,Rst);
LkReg     stg6      (l6[29:0],P6[4:0],Ps6[1:0],lk6[29:0],Pk6[4:0],ak6,rq7,
                               l5[29:0],P5[4:0],Ps5[1:0],lk5[29:0],Pk5[4:0],rq6,ak7,Rst);
LkReg     stg7      (l7[29:0],P7[4:0],Ps7[1:0],lk7[29:0],Pk7[4:0],ak7,Mrq,
                               l6[29:0],P6[4:0],Ps6[1:0],lk6[29:0],Pk6[4:0],rq7,Mak,Rst);

// the locked output inverters and write address multiplexers
Binv #30  inv1      (Lkd[29:0],    lk7[29:0]);
Binv #5   inv2      (LkP[4:0],     Pk7[4:0]);
mux2 #30  mux1      (Wa[29:0],     l7[29:0],l3[29:0],Wma);
mux2 #5   mux2      (WPa[4:0],     P7[4:0],P3[4:0],Wma);
mux2 #2   mux3      (Psel[1:0],    Ps7[1:0],Ps3[1:0],Wma);

// the write request steering logic
Select    sel2      (Wrqm,Wrqa,    Wrq,Wma,Rst);
Cgate2    C1        (Mgo,          Wrqm,Mrq,Rst);
Cgate2    C2        (Ago,          Wrqa,Arq,Rst);
Call      call      (Mak,Aak,Wgo,  Mgo,Ago,Wdn,Rst);

endmodule // LkFIFO


//   Address Interface
module AddInt (Wa0,Wa,APa0,PPr3,XPa0,XLr1,R15[31:2],LSMPa,MAr,MA[31:0],MAc[9:0],
                                             Wr0,W0[31:0],Wc0[9:0],APr0,A[31:0],LSMPr,Ntrm,
                                             dPC,PPa3,XPr0,XLa1,PCsel,MAa,Dabt1,Dabt0,Rst);

`include "AddInt.inc"

// Wc[] = SP, LSM, Ren, Wen, Usr, B/W, Opc, valid, destPC, PCpar
// MAc[] = Seq, Inc, Ren, Wen, Usr, B/W, Opc, valid, destPC, PCpar

// the W bus latch
EvtReg2 #(32,10) wlat    (W[31:0],Wc[9:0],Wa0,Wr,       W0[31:0],Wc0[9:0],Wr0,Wa,Rst);

DLand2    g0        (LSMpc,        Wc[8],dPC);
DLor2     g1        (dPCc,         Wc[1],LSMpc);
DLand2    g2        (LSin,         Wc[8],Ntrm);
DLor2     g3        (toInc,        LSin,Wc[3]);

// the A bus pipeline
Pipe2     apipe     (A2[31:0],APa0,APr2,     A[31:0],APr0,APa2,Rst);

// address bus and control bus muxs
mux4 #32  amux      (MA0[31:0],    PC[31:0],LS[31:0],A2[31:0],W[31:0],Adsl);
mux4 #10  cmux      (MAc0[9:0],    {pOpc,Vdd,Vdd,Vss,Usr,Vss,Vdd,Vdd,Vss,PCpar},
                                   {Vdd,Ntrm,MAc0[7],MAc0[6],Usr,Vss,Vss,Vdd,dPC,PCpar},
                                   {Vss,toInc,Wc[7:2],dPCc,Wc[0]},
                                   {Vss,toInc,Wc[7:2],dPCc,Wc[0]},Adsl);

EvtReg2 #(32,10) mar(MA[31:0],MAc[9:0],MAa0,MAr, MAz[31:0],MAc0[9:0],MAr0,MAa1,Rst);


// the address control logic
AddC      addC      (Wa,PCa1,APa2,LSa1,LSMPa,MAr0,PCpar,Usr,Adsl[1:0],
                         Wr,Wc[9],Wc[8],Wc[0],Wc[5],PCr2,APr2,LSr1,LSMPr,Ntrm,MAa0,Rst);


// a fudge to force the PC output to zero after reset
BNbuff #32 pcrst    (Nrst[31:0],   {32{Rst}});
Band #32  PCz       (MAz[31:0],    MA0[31:0],Nrst[31:0]);

// save previous Opc value
EvtLch    el2       (pOpc,         MAc0[3],MAa0,Rst);

// select incrementer bypass or -> PC or -> LSM reg
Select    s0        (Ninc,INCrin,  MAr,MAc[8],Rst);
Cgate2    c0        (MAa1,         MAa,x0,Rst);
DLxor2    xor0      (x0,           Ninc,x1);

Incre     inc       (Inc[31:0],    MA[31:0],INCrin,INCrout,x1,Rst);

Select    s1        (LSr0,PCr0,    INCrout,MAc[3],Rst);

EvtReg #32 pc0      (PCx[31:0],PCa0,PCr9,    Inc[31:0],PCr0,PCa9,Rst);
EvtReg #32 pc1      (PC[31:0],PCa9,PCr1,     PCx[31:0],PCr9,PCa2,Rst);
EvtReg #32 LSMr     (LS[31:0],LSa0,LSr1,     Inc[31:0],LSr0,LSa1,Rst);

DLxor2    xor1      (x1,           PCa0,LSa0);

// copy PC into PC pipe, reject 1st value and insert 1st event from Rst
Select    s2        (Z,PPr0,            PCr1,Z,Rst);
DLxor2    xor2      (x2,                PPa0,Z);
DLinv     inv       (NRst,              Rst);
Call      ca0       (nc0,PCa2,PCr2,     NRst,x2,PCa1,Rst);

// PC pipeline blocks
Pipe2     pcpipe    (pc3[31:0],PPa0,PPr3,    PC[31:0],PPr0,PPa3,Rst);
Pipe3     xpipe     (XP3[31:0],XPa0,XPr3,    pc3[31:0],XPr0,XPa3,Rst);
EvtReg #32 xlat     (XL[31:0],XLa0,XLr1,     XP3[31:0],XLr0,XLa1,Rst);
mux2 #32  pcmux     ({R15[31:2],nc1,nc2},    pc3[31:0],XL[31:0],PCsel);
```

```
Sink #3   sk1       ({nc0,nc1,nc2});

// Data Abort control of X pipe
DecWait2x1 dw0      (nDbt,XLr0,    XPr3,Dabt0,Dabt1,Rst);
DLxor2    x3        (XPa3,         nDbt,XLa0);

endmodule


module AddC (Wa,PCa,APa,LSMa,LSMPa,MARr,PCpar,Usr,Adsl[1:0],
                                   Wr,SP,LSM,PCpin,Usr0,PCr,APr,LSMr,LSMPr,Ntrm,MARa,Rst);

input Wr,SP,LSM,PCpin,Usr0,PCr,APr,LSMr,LSMPr,Ntrm,MARa,Rst;
output [1:0] Adsl;
output Wa, PCa, APa, LSMa, LSMPa, MARr, PCpar, Usr;

// Wr/a bundles SP (special load = LSRA or LSM), LSM, PCpin(parity in), Usr0
// LSMPr/a bundles Ntrm (= not last LSM cycle), dPC (= dest is PC)
// MARr/a bundles PCpar, Usr, Adsl[1:0],

// arbitrate into the PC loop
ArbitA    arb       (Wgo,PCgo,Wmux,     Wr,PCr,Wa,PCa,Rst);

// latch W sourced booleans
EvtLch    el0       (PCpar,        PCpin,Wgo,Rst);
EvtLch    el1       (Usr,          Usr0,Wgo,Rst);

// sort out the W sourced operations
Select    sl0       (r1,LSPr,      Wgo,SP,Rst);
Select    sl1       (LAr,LSMgo,    LSPr,LSM,Rst);
Cgate2    c0        (r0,           LAr,APr,Rst);
DLxor2    xor0      (x0,           APa,d1);
DLxor2    x2        (Wa,           x0,LSMx);

// the PC processing bit
Select    sl2       (r2,PCx,       PCgo,PCk,Rst);
DLxor2    x3        (PCk,          PCx,PCpar);
DLxor2    x4        (PCa,          PCx,d2);

// the LSM processing bit
DLxor2    xor5      (x5,           LSMgo,c2);
Cgate2    c1        (r3,           x5,LSMPr,Rst);
Select4   sl3       (s00,s01,s10,s11,   d3,{LSMmx,Ntrm},Rst);
DLxor2    x6        (LSMmx,        s01,s10);
DLxor2    x7        (LSMx,         s00,s10);
DLxor2    xor8      (x8,           s01,s11);
DLxor2    x9        (LSMa,         s11,s10);
DLxor2    x10       (LSMPa,        LSMx,x8);
Cgate2    mull2     (c2,           x8,LSMr,Rst);

// merge requests
Call4     call      (r0,r1,r2,r3,APa,d1,d2,d3,    MARr,MARa,Rst);

// produce the address mux controls
AddMxCtl_PLA #100 mxctl (Wmux,LSMmx,SP,LSM,  Adsl[1:0]);

endmodule // AddC
```

```
//   Data Interface
module DatInt (MARa,DOa,MEMr,MWR[31:0],MRRa,DWr,DW[31:0],DWusr,
               DWv,DWpc,INr1,IN1[31:0],flo[1:0],Imm[31:0],IMa0,IMr2,
                                              MARr,DOr,Db[31:0],DObw,MEMa,MRRr,MRR[31:0],
                                              Pabt,DWa,INa1,MADc[9:0],IMr0,IMa2,BigEnd,Rst);

`include "DatInt.inc"

//   MADc[] = byte[1:0], Ren, Wen, Usr, B/W, Opc, valid, destPC, PCpar
//   MCp[] = byte[1:0], Usr, B/W, Opc, valid, destPC, PCpar


EvtReg2 #(32,1) mrr (MDin[31:0],b1,MRRa,MDr, MRR[31:0],Pabt,MRRr,MDa,Rst);

Dout      dout      (MWR[31:0],DOa,MWRr,     Db[31:0],DObw,DOr,MWRa,Rst);

DstCtl    dstctl    (MDa,MCPa5,DIr,INr0,     MDr,MCPr5,MCP5[2],MCP5[3],DIa,INa0,Rst);

DLinv     inv       (Nabt,                   b1);
DLand2    g1        (DIv,                    Nabt,MCP5[2]);

Din       din       (DIa,DWr,DW[31:0],DWusr,DWv,DWpc,
                                   {MCP5[7:4],DIv,MCP5[1]},DIr,MDin[31:0],DWa,BigEnd,Rst);

Ipipe     ipipe     (IN1[31:0],flo[1:0],INr1,INa0,  MDin[31:0],{b1,MCP5[0]},INr0,INa1,Rst);

IMMpipe   imm       (Imm[31:0],IMa0,IMr2,    IN1[31:0],IMr0,IMa2,Rst);

MemCP     memCP     (MCP5[7:0],MCPa0,MCPr5,  {MADc[9:8],MADc[5:0]},MCPr0,MCPa5,Rst);

MemCtl    memCtl    (MWRa,MARa,MCPr0,MEMr,
                         MWRr,MARr,MADc[6],MADc[7],MADc[2],MADc[3],MADc[1],MCPa0,MEMa,Rst);

endmodule // DatInt


module Ipipe (i5[31:0],q5[1:0],r5,a0,        i0[31:0],q0[1:0],r0,a5,Rst);

input [31:0] i0; input [1:0] q0; input r0,a5,Rst;
output [31:0] i5; output [1:0] q5; output r5,a0;
wire [31:0] i1,i2,i3,i4; wire [1:0] q1,q2,q3,q4;

// Buffer pipeline: q[1:0] contains Prefetch abort (1) and PC parity (0) info

EvtReg2 #(32,2) lat0     (i1[31:0],q1[1:0],a0,r1, i0[31:0],q0[1:0],r0,a1,Rst);
EvtReg2 #(32,2) lat1     (i2[31:0],q2[1:0],a1,r2, i1[31:0],q1[1:0],r1,a2,Rst);
EvtReg2 #(32,2) lat2     (i3[31:0],q3[1:0],a2,r3, i2[31:0],q2[1:0],r2,a3,Rst);
EvtReg2 #(32,2) lat3     (i4[31:0],q4[1:0],a3,r4, i3[31:0],q3[1:0],r3,a4,Rst);
EvtReg2 #(32,2) lat4     (i5[31:0],q5[1:0],a4,r5, i4[31:0],q4[1:0],r4,a5,Rst);

endmodule // Ipipe


module Din (a0,r1,D1[31:0],D1u,D1v,D1pc,     b0[5:0],r0,D0[31:0],a1,BigEnd,Rst);

input [31:0] D0; input [5:0] b0; input r0,a1,BigEnd,Rst;
output [31:0] D1; output a0,r1,D1u,D1v,D1pc;
wire [31:0] D0a;
```

```
// the data input block, including byte selection and alignment logic
// b0[5:0] contains: byte[1:0]; Usr; Byte/Word; Valid; PCdest;

Sink #1   sk        (BigEnd);
Din_blk   din       (D0a[31:0],    D0[31:0],b0[5:4],b0[2]);
EvtReg2 #(32,3) lat0 (D1[31:0],{D1u,D1v,D1pc},a0,r1,   D0a[31:0],{b0[3],b0[1:0]},r0,a1,Rst);
endmodule // Din


module Dout (D3[31:0],a0,r3,  D0[31:0],bw,r0,a3,Rst);

input [31:0] D0; input bw,r0,a3,Rst;
output [31:0] D3; output a0,r3;
wire [31:0] D1,D1a,D2;

// the data out block, including byte replication logic

EvtReg2 #(32,1) lat0     (D1[31:0],b1,a0,r1, D0[31:0],bw,r0,a1,Rst);
Dout_blk       dout      (D1a[31:0],         D1[31:0],b1);
EvtReg #32     lat1      (D2[31:0],a1,r2,    D1a[31:0],r1,a2,Rst);
EvtReg #32     lat2      (D3[31:0],a2,r3,    D2[31:0],r2,a3,Rst);

endmodule // Dout


module DstCtl (MMa,MCa,DIr,IPr,MMr,MCr,Val,Opc,DIa,IPa,Rst);

input MMr,MCr,Val,Opc,DIa,IPa,Rst;
output MMa,MCa,DIr,IPr;

// read data destination control: sends the incoming data to Din or Ipipe
// - also sets Din up early to receive incoming data

Select    s0        (Dat,Op,       MCr,Opc,Rst);
Select    s1        (nVd,Vd,       Dat,Val,Rst);
DecWait2x1 dw       (IPr,Vd1,      MMr,Op,Vd,Rst);
Call      ca0       (b0,b1,DIr,    nVd,Vd1,DIa,Rst);
DLxor2    x0        (MMa,          IPa,b1);
DLxor2    x1        (MCa,          MMa,b0);

endmodule // DstCtl


module MemCP (d5[7:0],a0,r5,       d0[7:0],r0,a5,Rst);

input [7:0] d0; input r0,a5,Rst;
output [7:0] d5; output a0,r5;
wire [7:0] d1,d2,d3,d4;

// the memory control pipeline
//   MCp[] = byte[1:0], Usr, b/w, Opc, valid, destPC, PCpar

EvtReg #8 l0        (d1[7:0],a0,r1,     d0[7:0],r0,a1,Rst);
EvtReg #8 l1        (d2[7:0],a1,r2,     d1[7:0],r1,a2,Rst);
EvtReg #8 l2        (d3[7:0],a2,r3,     d2[7:0],r2,a3,Rst);
EvtReg #8 l3        (d4[7:0],a3,r4,     d3[7:0],r3,a4,Rst);
EvtReg #8 l4        (d5[7:0],a4,r5,     d4[7:0],r4,a5,Rst);

endmodule // MemCP
```

```
module MemCtl (Da,MAa,MCr,MMr,     Dr,MAr,Wen,Ren,Val,Opc,dPC,MCa,MMa,Rst);

input Dr,MAr,Wen,Ren,Val,Opc,dPC,MCa,MMa,Rst;
output Da,MAa,MCr,MMr;

// the memory control block

Select    s0        (noW,W,        MAr,Wen,Rst);
Cgate2    mull0     (c0,           W,Dr,Rst);
Call      call      (Da,Ma0,ca,    c0,noW,c1,Rst);
DLxor2    x0        (MAa,          Ma0,Da);

DLor2     g0        (NReg,         Opc,dPC);
DLinv     inv       (Reg,          NReg);
DLor2     g1        (DoIt,         Reg,Val);
DLand2    g2        (MCen,         DoIt,Ren);

Select    s1        (noR,MCr,      ca,MCen,Rst);
Select    s2        (noM,MMr,      ca,Val,Rst);
DLxor2    xor1      (x1,           noR,MCa);
DLxor2    xor2      (x2,           noM,MMa);
Cgate2    mull1     (c1,           x1,x2,Rst);

endmodule // MemCtl


module IMMpipe (i2[31:0],a0,r2,    i0[31:0],r0,a2,Rst);

input [31:0] i0; input r0,a2,Rst;
output [31:0] i2; output a0,r2;
wire [31:0] i1,j1;

// the immediate field extraction and pipeline:

EvtReg #32 lat0     (i1[31:0],a0,r1,    i0[31:0],r0,a1,Rst);
ImmExt     imm      (j1[31:0],          i1[31:0]);
EvtReg #32 lat1     (i2[31:0],a1,r2,    j1[31:0],r1,a2,Rst);

endmodule // Ipipe


// Micropipelined ARM Execution Pipe
module ExecP (RDa,C2a,C3a,NGa2,SHe,IMa2,PCpar,ALUgo,ALUok,mode[5:0],psrC,
              WRr,W[31:0],Wc[9:0],Wq[1:0],APr,DOr,nMLb[31:0],RSr,Dabt0,
                                    RDr,Na[31:0],Nb[31:0],C2r,c2[7:0],C3r,c3[22:0],vec3[2:0],
                                    ALfs[9:0],NGr2,ng[5:0],Sh[31:0],ShC,SHd,IMr2,Imd[6:0],
                                    DW[31:0],DWv,DWusr,WRa,Wsel,APa,DOa,RSa,Dabt[1:0],Rst);

`include "ExecP.inc"

Ctl2      ctl2      (C2a,APr,DOr,RDa,IMa2,OPr0,MLe,SHe,CPr0,NGa2,RSr,
                         C2r,c2[7:0],APa,DOa,RDr,IMr2,OPa0,MLd,SHd,CPa0,NGr2,RSa,ALx,Rst);

Ctl3      ctl3      (C3a,CPa1,ALe,ALel,OPa1,WLr,alP[12:0],
                     PCp1,PCpar,ALUgo,CPmx[3:0],Dabt0,ALx,
                              C3r,c3[22:0],CPr1,ALd,OPr1,WLa,Dabt[1:0],Pass,Usr0,Usr,Rst);
```

```
ExecDP    excDP     (MLd,nMLb[31:0],OPa0,OPr1,CPa0,Pass,CPr1,psrC,
                     mode[5:0],Usr0,Usr,ALd,W[31:0],Wc[9:0],Wq[1:0],WLa,WRr,
                            Na[31:0],Nb[31:0],MLe,c2[1],c2[2],ng[5:0],Sh[31:0],OPr0,OPa1,
                            CPr0,ShC,ALe,ALe1,CPmx[3:0],Imd[6:0],vec3[2:0],CPa1,PCp1,
                            ALfs[9:0],DW[31:0],DWv,DWusr,alP[12:0],WLr,PCpar,WRa,Wsel,Rst);

DLbuff    bf0       (ALUok,        alP[2]);

endmodule // ExecP


//   Control 2 decode logic
module Ctl2 (C2a,APr,DOr,RDa,IMa,OPr,MLe,SHe,CPr,NGa,RSr,
                              C2r,c2[7:0],APa,DOa,RDr,IMr,OPa,MLd,SHd,CPa,NGr,RSa,ALx,Rst);

input [7:0] c2;
input C2r,APa,DOa,RDr,IMr,OPa,MLd,SHd,CPa,NGr,RSa,ALx,Rst;
output C2a,APr,DOr,RDa,IMa,OPr,MLe,SHe,CPr,NGa,RSr;

//   c2[] = toI3, ~Rs, cpCP, ~toDO, ~toA, nGen, ~Mult, NImm

// sync with appropriate shifter input
Select    sl0       (ShIm,C2r1,    C2r,c2[0],Rst);          // select Imm or Reg operand
DecWait2x1 dw0      (C2r2,RDx,     RDr,C2r1,IMa,Rst);       // sync reg on shift
Cgate2    c1        (SHg1,         ShIm,IMr,Rst);

// bypass for cycles which terminate here
Select    sl1       (byP,SHg0,     C2r2,c2[7],Rst);
DLxor2    x0        (NGx1,         NGx,byP);

// activate multiplier if required
Select    sl2       (MLe,nML,      SHg0,c2[1],Rst);
DLxor2    x1        (SHg2,         nML,MLd);

// merge calls and activate shifter
Call      ca0       (IMa,SHx0,SHgo, SHg1,SHg2,SHr,Rst);
DLinv     inv1      (nSHa,         SHa);
DLinv     inv2      (nALx,         ALx);
Cgate3    mull2     (SHst,         SHgo,nSHa,nALx,Rst);
DLxor2    x2        (SHe,          SHst,CPx1);
Toggle    t0        (SHr,SHa,      SHd,Rst);
DLxor2    x3        (SHx,          RDx,SHx0);

// accept nGen input
DLxor2    x40       (NG0,          SHg0,ShIm);
Select    sl3       (noNG,NG,      NG0,c2[2],Rst);
Cgate2    c3        (NG1,          NG,NGr,Rst);
Call      ca1       (NGa,NG2,NGgo, NG1,noNG,CPx1,Rst);
DLxor2    x4        (NGx,          NG2,NGa);

// other pipelines
Cgate2    c4        (RDgo,         C2r,RDr,Rst);

// the A pipe
Select    sl4       (APr,noA,      RDgo,c2[3],Rst);
DLxor2    x5        (APx,          APa,noA);

// the data out pipe
Select    sl5       (DOr,noDO,     RDgo,c2[4],Rst);
DLxor2    x6        (DOx,          DOa,noDO);

// the CPSR copy latch( SHr used to capture ShC )
Select    sl6       (noCP,CPr,     SHr,c2[5],Rst);
DLxor2    x7        (CPx,          CPa,noCP);
Cgate2    c5        (CPx1,         CPx,OPa,Rst);

// the Rs latch
Select    sl7       (RSr,noRS,     RDgo,c2[6],Rst);
DLxor2    x8        (RSx,          RSa,noRS);

// call operand latch
Cgate2    c6        (OPr,          SHx,NGgo,Rst);

// wait for all outputs to complete
Cgate2    c7        (apdo,         APx,DOx,Rst);
Cgate2    c8        (rsng,         NGx1,RSx,Rst);
Cgate2    c9        (C2a,          apdo,rsng,Rst);
DLbuff    x9        (RDa,          C2a);

endmodule // Ctl2


//   Control 3 decode logic
module Ctl3 (C3a,CPa1,ALe,ALe1,OPa,WLr,alP[12:0],PCp1,PCpar,ALUgo,CPmx[3:0],Dabt0,ALx,
                                  C3r,c3[22:0],CPr1,ALd,OPr,WLa,Dabt[1:0],Pass,Usr0,Usr,Rst);
input [22:0] c3;
input [1:0] Dabt;
input C3r,CPr1,ALd,OPr,WLa,Pass,Usr0,Usr,Rst;

output [12:0] alP;
output [3:0] CPmx;
output C3a,CPa1,ALe,ALe1,OPa,WLr,PCp1,PCpar,ALUgo,Dabt0,ALx;

//   c3[] = UseCP,S,F,C,wCP[2:0],Ral,Rcnd,~ALUwt,~DabtWt,
//                                tPCp[1:0],Wreg,Wadd,SP,LSM,Ren,Wen,B/W,Opc,destPC,Rsel
//   alP[] = Rsel,Wreg,Wadd,SP,LSM,Ren,Wen,Usr,B/W,Opc,valid,destPC,PCpar

// the CPSR mux control
CPmxC_PLA #100 dec0 (Rst,Pass,Usr0,c3[21:16],     CPmx[3:0]);

// use CPSR copy?
Cgate2    c0        (ALr0,         C3r,OPr,Rst);
DLinv     inv0      (Fail,         Pass);
DLor2     g1        (UseCP,        Fail,c3[22]);
Select    sel0      (nUcp,Ucp,     ALr0,UseCP,Rst);
Cgate2    c1        (Ucp1,         CPr1,Ucp,Rst);
Call      ca0       (C3a1,CPa1,ALr,nUcp,Ucp1,OPa,Rst);
DLxor2    x0        (C3a,          C3a1,CPa1);

// the ALU dynamic control - (c3[18] && c3[17] decodes LDM cycle 4)
DLinv     inv1      (nALrdy,       ALrdy);
DLinv     inv2      (nWLa1,        WLa1);
Cgate3    c2        (ALgo,         ALr,nALrdy,nWLa1,Rst);
DLor2     g2        (nLDM4,        c3[18],c3[17]);
DLinv     inv3      (LDM4,         nLDM4);
DLand2    g3        (nALdo,        DP,LDM4);
DLinv     inv4      (ALdo,         nALdo);
Cgate2    mull3     (ALx1,         ALx,tPCp0x,Rst);
```

```
DLxor2    x1        (ALe,          ALgo,ALx1);
Toggle    t0        (ALx,ALrdy,    ALd,Rst);
DLand2    g31       (ALel,         ALdo,ALe);

// construct the control bus
DLor2     g5        (v0,           nLDM4,DP);
DLand2    g6        (Val,          v0,Pass);

LatchR #13 lat      (alP[12:0],    {c3[0],c3[9:4],Usr,c3[3:2],Val,c3[1],PCpar},ALe,Rst);

// pass on the result if needed
DLand2    g7        (g0,           c3[14],Pass);
DLor2     g8        (WLok,         c3[15],g0);
Select    sel2      (WLby,WLr,     ALx1,WLok,Rst);
DLxor2    x3        (WLa1,         WLby,WLa);

// toggle PCpar ?
DLand2    g9        (tPCp0,        c3[10],Pass);
Select    sel3      (nPCt0,PCt0,   ALr,tPCp0,Rst);
Select    sel30     (nPCd0,PCd0,   PCt0,DP,Rst);
DLxor2    x40       (DP,           DB1,PCd0);
DLxor2    x41       (PCdx,         nPCd0,PCd0);
DLxor2    x4        (tPCp0x,       nPCt0,PCdx);
DLxor2    x5        (tog0,         DB1,nPCd0);
DLinv     g90       (nPCp1,        PCp1);
EvtLch    elat0     (PCpar,        nPCp1,tog0,Rst);

// toggle PCp1 ?
Cgate2    c4        (tPCxC,        tPCp0x,DBx,Rst);
Select    sel4      (nPCt1,PCt1,   tPCxC,c3[11],Rst);
DLxor2    x6        (tPCp1x,       PCt1,nPCt1);
EvtLch    elat1     (PCp1,         PCpar,PCt1,Rst);

// signal completion to Dec1 if needed
Select    sel6      (ALUgo,nALg,   ALx,c3[13],Rst);
DLxor2    x8        (ALgx,         ALUgo,nALg);

// wait for abort response if needed
Select    sel7      (DBw0,nDBw,    ALr,c3[12],Rst);
Select    sel8      (nDB,DBw,      DBw0,Pass,Rst);
DecWait2x1 dw0      (DB0,DB1,      DBw,Dabt[0],Dabt[1],Rst);
DLxor2    x9        (DB2,          DB1,DB0);
DLxor2    x10       (DB3,          nDB,DB2);
DLxor2    x11       (DBx,          DB3,nDBw);
DLxor2    x12       (Dabt0,        nDB,Dabt[0]);

// gather completion signals
Cgate2    c5        (OPa,          ALgx,tPCp1x,Rst);

endmodule // Ctl3


// Micropipelined ARM Execution Datapath
module ExecDP (MLd,nMLb[31:0],OPa0,OPr1,CPa0,Pass,CPr1,psrC,mode[5:0],
               Usr0,Usr,ALd,W[31:0],Wc[9:0],Wq[1:0],WLa,WRr,
                         Na[31:0],Nb[31:0],MLe,MLmx,nGen,ng[5:0],Sh[31:0],OPr0,OPa1,
                         CPr0,ShC,ALe,ALel,CPmx[3:0],Imd[6:0],vec3[2:0],CPa1,PCp1,
                         ALfs[9:0],DW[31:0],DWv,DWusr,alP[12:0],WLr,PCpar,WRa,Wsel,Rst);

`include "ExecDP.inc"

// the 2nd execution stage (Reg is the 1st stage)
Mult      mul       (MLa[31:0],nMLb[31:0],MLd,    Na[31:0],Nb[31:0],MLe,MLmx,Rst);

mux2 #32  opAmx     (OPaM[31:0],   MLa[31:0],{{26{Vss}},ng[5:0]},nGen);

EvtReg2 #(32,32) oplat (OPa[31:0],OPb[31:0],OPa0,OPr1, OPaM[31:0],Sh[31:0],OPr0,OPa1,Rst);

// the CPSR bridges the 2nd & 3rd execution stages
CPSR      cpsr      (CPa0,ShC1,Pass,CPr1,psrC,aluC,aluV,mode[5:0],CP1[10:0],Usr0,Usr,
                                   CPr0,ShC,ALe,{alu[31:28],alu[7:6],alu[4:0]},ALfg[3:0],
                                   CPmx[3:0],Imd[6:0],vec3[2:0],CPa1,PCp1,Rst);

// 3rd execution stage
ALU       aluX      (alu0[31:0],ALfg[3:0],ALd,
                                        OPa[31:0],OPb[31:0],ALfs[9:0],aluC,aluV,ShC1,ALe);

LatchR #32 alulat   (alu[31:0],    alu0[31:0],ALel,Rst);

// the result stage
mux2 #32  rmux      (RE[31:0],
                         alu[31:0],{CP1[10:7],{20{Vss}},CP1[6:5],Vss,CP1[4:0]},alP[12]);

EvtReg2 #(32,12) rlat (WR[31:0],{Wq[1:0],WRq[9:0]},WLa,WRr, RE[31:0],alP[11:0],WLr,WRa,Rst);

//   alP[12] = Rsel,
//   WRq[] = SP, LSM, Ren, Wen, Usr, B/W, Opc, valid, destPC, PCpar
//   Wq[] = Wreg, Wadd

// write bus control
//   Wc[] = SP, LSM, Ren, Wen, Usr, B/W, Opc, valid, destPC, PCpar
mux2 #32  wmux      (W[31:0],      DW[31:0],WR[31:0],Wsel);
mux2 #10  wcmx      (Wc[9:0],      {Vss,Vss,Vdd,Vss,DWusr,Vss,Vdd,
                                    DWv,Vss,PCpar},WRq[9:0],Wsel);

endmodule // ExecDP


//   CPSR
module CPSR (CPa0,ShC1,Pass,CPr1,psrC,aluC,aluV,mode[5:0],CP1[10:0],Usr0,Usr,
                                             CPr0,ShC0,ALe,alu[10:0],ALfg[3:0],CPmx[3:0],
                                             Imd[6:0],vec3[2:0],CPa1,PCp1,Rst);

`include "CPSR.inc"

//   CP0[] = aluN, aluZ, aluC, aluV, Omd[6:0]
//   CP3[] = psrN, psrZ, psrC, psrV, mode[5:4], modeNC, mode[3:0]

// check the condition codes and PC parity
Cond_PLA #100 cond  (Imd[6:0],PCp1,psrN,psrZ,psrC,psrV,     Pass0);

// latch the old CPSR and the new Shift carry and condition pass/fail
EvtReg #13     reg0 ({ShC1,Pass,aluN,aluZ,aluC,aluV,Omd[6:0]},CPa0,CPr1,
                                             {ShC0,Pass0,psrN,psrZ,psrC,psrV,mode[5:4],
                                              modeNC,mode[3:0]},CPr0,CPa1,Rst);
LatchR #11     lat0 (CP1[10:0],    {aluN,aluZ,aluC,aluV,Omd[6:0]},ALe,Rst);
```

```
// evaluate the new CPSR values
Mode_PLA #100 mode0 (vec3[2:0],Omd[5],  modd[6:0]);
mux4 #4   mx1       (CP2[10:7],    {aluN,aluZ,aluC,aluV},alu[10:7],ALfg[3:0],
                                   {Vss,Vss,Vss,Vss}, CPmx[3:2]);
mux4 #7   mx2       (CP2[6:0],     Omd[6:0],alu[6:0],modd[6:0],
                                   {Vdd,Vdd,Vdd,Vss,Vss,Vdd,Vdd},CPmx[1:0]);

// reset sets I, F and SVC mode
DLor4     sg00      (cpm0,         CPmx[0],CPmx[1],CPmx[2],CPmx[3]);
DLand2    sg01      (ALe0,         ALe,cpm0);
DLor2     sg1       (ALe1,         ALe0,Rst);

// latch synchronously with ALU output
Latch #11 lat1      ({psrN,psrZ,psrC,psrV,mode[5:4],modeNC,mode[3:0]},CP2[10:0],ALe1);

// were we, and are we now, in a user mode ?
DLor2     sg2       (NUsr0,        CP1[1],CP1[0]);
DLinv     inv2      (Usr0,         NUsr0);
DLor2     sg3       (NUsr,         CP2[1],CP2[0]);
DLinv     inv3      (Usr,          NUsr);

endmodule // CPSR


//   Write Bus Control
module WrCtl (Da,Wa,RWr,ADr,Wmux,  Dr,destPC,Val,Wr,Wc[1:0],RWa,ADa,WLx,Rst);

input [1:0] Wc;
input Dr,destPC,Val,Wr,RWa,ADa,WLx,Rst;
output Da,Wa,RWr,ADr,Wmux;

//   Wc[] = Wreg, Wadd

ArbitA    arb       (Dgo,Wgo,nWmux,     Dr,Wr1,Da,Wa,Rst);
DLinv     inv0      (Wmux,         nWmux);
Select    s0        (Drg,Dpc0,     Dgo,destPC,Rst);
Select    s1        (v0,Dpc,       Dpc0,Val,Rst);
Select    s2        (Npc,Wpc,      Wgo,Wc[0],Rst);
Select    s3        (Nrg,Wrg,      Wgo,Wc[1],Rst);
Call      ca0       (Dra,Wra,RWr,  Drg,Wrg,RWa,Rst);
Call      ca1       (Dpa,Wpa,ADr,  Dpc,Wpc,ADa,Rst);
DLxor2    xr0       (Da0,          Dra,Dpa);
DLxor2    xr1       (x1,           Wra,Nrg);
DLxor2    xr2       (x2,           Wpa,Npc);
Cgate2    c1        (Wa,           x1,x2,Rst);

//W bus deadlock avoidance

Select    s4        (nWA,WAr,      Wr,Wc[0],Rst);
Cgate2    c0        (WAok,         WAr,Lok,Rst);
DLinv     g0        (Lok,          Wd1);
DLxor2    xr3       (Wr1,          nWA,WAok);
DecWait2x1 dw0      (Dd1,Wd1,      dwf,Dpc,Wpc,Rst);
DLxor2    xr4       (x4,           Dd1,Wd1);
Cgate2    c2        (dwf,          WLx,Nx4,Rst);
DLinv     inv1      (Nx4,          x4);
DLxor2    xr6       (Da,           Da0,v0);

endmodule // WrCtl
```

```
// NGen micropipeline stage
module NGen (NGa0,NGr2,ng[5:0],    NGr0,IN[15:0],nGn[1:0],vect[2:0],NGa2,Rst);

`include "NGen.inc"

EvtReg #21 l0       (ot[20:0],NGa0,NGr1,     {IN[15:0],nGn[1:0],vect[2:0]},NGr0,NGa1,Rst);

Select    sl0       (noNG,Nr0,     NGr1,ot[4],Rst);
NGenX     ngen      (Nr1,n[3:0],   Nr0,ot[20:5],Rst);
DLxor2    x0        (NGr1a,        Nr1,noNG);
mux4 #6   mx        (ng0[5:0],     {Vss,Vss,Vss,Vss,Vdd,Vdd},
                                   {Vss,ot[2:0],Vss,Vss},
                                   {n[3:0],Vss,Vss},
                                   {n[3:0],Vdd,Vdd}, ot[4:3]);

EvtReg #6 l1        (ng[5:0],NGa1,NGr2,      ng0[5:0],NGr1a,NGa2,Rst);

endmodule // NGen
```

```
//**************************************************************************************//
//                                                                                      //
//    The following file contains the Behavioural models of miscellaneous ARM functions   //
//                                                                                      //
//**************************************************************************************//
`timescale 10ps /10ps

//**********************************************************************//
//                   Arithmetic Logic Unit (ALU)                         //
//**********************************************************************//
module ALU(alu[31:0],flags[3:0],rout,   ain[31:0],bin[31:0],func[9:0],psrC,psrV,shC,rin);

`define ALU_Add 800
`define ALU_Logic 400
`define ALU_Recover 300

output [31:0] alu;
reg [31:0] alu;
output [3:0] flags;
reg [3:0] flags;
output rout;
reg rout;
input [31:0] ain, bin;
input [9:0] func;
input psrC,psrV,shC,rin;
integer A, B, out;
reg [31:0] delay;
reg [9:0] f;
reg [3:0] nzcv;
reg c0,c1,v1;

always @(posedge rin)
     begin
          delay = `ALU_Logic;
          A = 0; B = 0; c1 = shC; v1 = psrV; f = ~func; nzcv = 0;

          if (f & `h80) A = ain;
     else if (f & `h40) A = ~ain;

          if (f & `h20) B = bin;
     else if (f & `h10) B = ~bin;

     case (f & `hF)
          4'h1:     out = A ^ B;                  // A xor B
          4'h2:     out = A | B;                  // A or B
          4'h4:     out = A & B;                  // A and B
          4'h8:     begin                         // A + B
                         case (f & `h300)// carry in logic
                              10'h000:  c0 = 1;
                              10'h200:  c0 = psrC;
                              10'h300:  c0 = 0;
                              default:  begin
                                             $display("Illegal carry select in ALU\n");
                                             $stop(1);
                                        end
                         endcase

                         out = A + B + c0;
                         c1 = (A<0 && B<0) || (A<0 && out>=0) || (B<0 && out>=0);
                         v1 = (A<0 && B<0 && out>=0) || (A>=0 && B>=0 && out<0);
                         delay = `ALU_Add;
                    end
          default:  begin
                         $display("Illegal ALU function code\n");
                         $display("Current simulation time is %d\n", $time);
                          out = ~0;
                    end
     endcase

     nzcv = ((out<0)<<3) + ((out==0)<<2) + (c1<<1) + v1;

     #(delay) alu = out;
     flags = nzcv;
     #100 rout = 1;
     end

always @(negedge rin)
     fork
          #100 alu = `bx;
          #100 flags = `bx;
          #(`ALU_Recover) rout = 0;
     join

endmodule


//**********************************************************************//
//                              Multiplier                              //
//**********************************************************************//
module Mult(outA[31:0],outB[31:0],rout, inA[31:0],inB[31:0],rin,Mmux,rst);

`define MultDin_Dout 27
`define MultMux_Dout 43
`define MultRst_Rout 47
`define MultCompute 2500

output [31:0] outA, outB;
reg [31:0] outA, outB;
output rout;
reg rout;
input [31:0] inA, inB;
input rin, Mmux, rst;
reg [31:0] sum, carry;

`define Mult_reset #(`MultRst_Rout) rout=0;
`define Mult_undef begin outA='bx; outB='bx; rout=1'bx; end


always @(rst)
     case(rst)
          1'bx: `Mult_undef
          1'b1: `Mult_reset
          1'b0: begin
                    #(`MultDin_Dout) outA = inA;
                    outB = inB;
               end
     endcase
```

```
always @(Mmux)
     case(Mmux)
          1'bx:begin
                    #(`MultMux_Dout) outA = 'bx;
                    outB = 'bx;
               end
          1'b1:begin
                    #(`MultMux_Dout) outA = inA;
                    outB = inB;
               end
          1'b0:begin
                    #(`MultMux_Dout) outA = 'bx;
                    outB = 'bx;
               end
     endcase

always @(inA)
     if (!rst)
          #(`MultDin_Dout) outA = inA;

always @(inB)
     if (!rst)
          #(`MultDin_Dout) outB = inB;

always @(rin)
     begin
          if (!rst)
               fork
                    begin
                         #(`MultDin_Dout) outA = 'bx;
                         outB = 'bx;
                    end
                    begin
                         // SBF's method of generating non-trivial partial sum
                         // and carry, which add to the correct answer.
                         sum = inA * inB;
                         carry = sum >> 1;
                         sum = sum - carry;
                         #(`MultCompute) outA = sum;
                         outB = carry;
                         #(100) rout = rin;
                    end
               join
     end

endmodule


//**********************************************************************//
//        Shifter
//**********************************************************************//
module Shift(Sh[31:0],ShC,ShD,Mlt[31:0],Imm[31:0],sel,op[9:0],Cin,ShE);

`define ShCompute 450
`define ShRecover 100

output [31:0] Sh;
reg [31:0] Sh;
output ShC,ShD;
reg ShC,ShD;
input [31:0] Mlt,Imm;
input [9:0] op;
input sel,Cin,ShE;

reg [31:0] val,mask,ones;
reg [4:0] n,m;
reg carry,b0,b31;

initial
     ones = 32'hFFFFFFFF;

always @(posedge ShE)
     begin
          val = sel ? Mlt : Imm;
          b0 = val & 1'b1;
          b31 = (val>>31) & 1'b1;
          n = op & 5'h1F;// 0,1...31
          m = (n-1) & 5'h1F;// 31,0...30
          carry = (val>>m) & 1'b1;// carry for right shift

          case((op>>5) & 3)
               2'b00:    begin
                              carry = (n ? (val>>(32-n)) & 1 : Cin);       // LSL
                              val = val<<n;
                         end
               2'b01:    val = (n ? (val>>n) : 0);                         // LSR
               2'b10:    if (val[31]==1)                                   // ASR
                              begin
                                   mask = (ones<<(32-n));
                                   val = (n ? ((val>>n) | mask) : ones);
                              end
                         else
                              val = (n ? (val>>n) : 0);
               2'b11:    if (n>0)
                              val = (val>>n) | (val<<(32-n));              // ROR
                         else
                              begin
                                   carry = b0;
                                   val = (Cin<<31) | (val>>1);             // RRX
                              end
               default:  $display("Illegal SHIFTx operation code !!");
          endcase

          case((op>>7) & 7)
               3'b110:   ;
               3'b111:   carry = 0;
               3'b101:   carry = b0;
               3'b011:   carry = b31;
               default:  $display("Illegal SHIFTx carry code: %d!!", ((op>>7) & 7));
          endcase

          #(`ShCompute) Sh = val;
          ShC = carry;
          #50 ShD = 1;
     end
```

```
always @(negedge ShE)
     begin
          #(`ShRecover) Sh = `bx;
          ShC = `bx;
          ShD = 0;
     end

endmodule


// Contents of the Register Bank and associated functions library.
//
// WriteEn(enW[29:0],psWm[4:0],psWf[4:0], Wa[29:0],WPa[4:0],Psel[1:0],Wen);
// WrComp(Wcmp, Wen,enW[29:0],psWm[4:0],psWf[4:0]);
// Decode(dcA[30:0],dcP[4:0], A[3:0],NBS[6:0],en,rNp);
// REG(Na[31:0],Nb[31:0],Adn,Bdn,       W[31:0],nPC[31:2],enA[30:0],psA[4:0],
//                                      enB[30:0],enW[29:0],psWm[4:0],psWf[4:0],Ren);

//********************************************************************//
//                  REGISTER BANK (Event Driven)                     //
//********************************************************************//
module REG(Na,Nb,Adn,Bdn,     W,nPC,enA,psA,enB,enW,psWm,psWf,Ren);

`define REGrd_delay 500
`define REGdn_delay 100


output [31:0] Na,Nb;
reg [31:0] Na,Nb;
output Adn,Bdn;
reg Adn,Bdn;
input [31:0] W;
input [31:2] nPC;
input [30:0] enA,enB;
input [29:0] enW;
input [4:0] psA,psWm,psWf;
input Ren;
reg [31:0] Rbnk[0:36];
reg [31:0] Rprev[0:36];
reg [31:0] Mtime[0:36];
reg [31:0] j;
reg [5:0] rdA,rdPS,rdB,wrW,wrPSm,wrPSf;


function [5:0] un2bin;                         // convert unary representation to binary
input [31:0] unary;
reg [5:0] bin,index;
     begin
          bin = 6'h3F;
          for (index=0; index<32; index=index+1)
               if (unary & (1<<index))
                    bin = index;
               un2bin = bin;
     end
endfunction

initial
     begin
          for (j=0; j<37; j=j+1)
               begin
                    Rbnk[j] = 0;
                    Rprev[j] = 0;
                    Mtime[j] = 0;
               end
          #(`REGdn_delay) Adn = 0; Bdn = 0;
     end

always @(nPC)                                  // update PC
     begin
          Rprev[30] = Rbnk[30];
          Mtime[30] = $stime;
          Rbnk[30] = (nPC<<2);
     end

always @(enA)                                  // register read A
     begin
          rdA = un2bin(enA);
          if (rdA!=6'h3F)
               begin
                    #(`REGrd_delay) Na = Rbnk[rdA];
                    Adn = 1;
               end
     end

always @(psA)                                  // register read PSR
     begin
          rdPS = un2bin(psA);
          if (rdPS!=6'h3F)
               begin
                    #(`REGrd_delay) Na = Rbnk[rdPS+32];
                    Adn = 1;
               end
     end

always @(enB)                                  // register read B
     begin
          rdB = un2bin(enB);
          if (rdB!=6'h3F)
               begin
                    #(`REGrd_delay) Nb = Rbnk[rdB];
                    Bdn = 1;
               end
     end

always @(negedge Ren)                          // precharge data buses
     begin
          #(`REGdn_delay) Na = `bx;
          Nb = `bx;
          Adn = 0;
          Bdn = 0;
     end

always @(enW)                                  // register write
     begin
          wrW = un2bin(enW);
```

```
            if (wrW!=6'h3F)
                begin
                    Rprev[wrW] = Rbnk[wrW];
                    Mtime[wrW] = $stime;
                    Rbnk[wrW] = W;
                end
        end

always @(psWm)                          // register write PSR mode
        begin
            wrPSm = un2bin(psWm);
            if (wrPSm!=6'h3F)
                begin
                    Rprev[wrPSm+32] = Rbnk[wrPSm+32];
                    Mtime[wrPSm+32] = $stime;
                    Rbnk[wrPSm+32] = (W & 32'hDF) | (Rbnk[wrPSm+32] & ~32'hDF);
                end
        end

always @(psWf)                          // register write PSR flags
        begin
            wrPSf = un2bin(psWf);
            if (wrPSf!=6'h3F)
                begin
                    Rprev[wrPSf+32] = Rbnk[wrPSf+32];
                    Mtime[wrPSf+32] = $stime;
                    Rbnk[wrPSf+32] = (W & 32'hF0000000) | (Rbnk[wrPSf+32] & ~32'hF0000000);
                end
        end

endmodule // REG


//*******************************************************************//
//                      Write Enable Logic                           //
//*******************************************************************//
module WriteEn(enW,psWm,psWf,      Wa,WPa,Psel,Wen);

`define WriteEn_delay 200

output [29:0] enW;
reg [29:0] enW;
output [4:0] psWm,psWf;
reg [4:0] psWm,psWf;
input [29:0] Wa;
input [4:0] WPa;
input [1:0] Psel;
input Wen;

always @(Wa or WPa or Psel or Wen)
        fork
            if (Wen)
                #(`WriteEn_delay) enW = Wa;
            else
                #(`WriteEn_delay) enW = 0;
            if (Wen && (Psel&2))
                #(`WriteEn_delay) psWf = WPa;
            else
                #(`WriteEn_delay) psWf = 0;
            if (Wen && (Psel&1))
                #(`WriteEn_delay) psWm = WPa;
            else
                #(`WriteEn_delay) psWm = 0;
        join

endmodule // WriteEn


//*******************************************************************//
//                      Write Completion Logic                       //
//*******************************************************************//
module WrComp(Wcmp,      Wen,enW,psWm,psWf);

`define WrComp_delay 300

output Wcmp;
reg Wcmp;
input [29:0] enW;
input [4:0] psWm,psWf;
input Wen;
reg [29:0] w;
reg comp;


initial
        Wcmp = 0;

always @(Wen or enW or psWm or psWf)
        begin
            w = enW | psWm | psWf;
            comp = Wen && !(w==0);
            if (Wcmp!=comp)
                #(`WrComp_delay) Wcmp = comp;
        end

endmodule // WrComp


//*******************************************************************//
//                      Address Decode Logic                         //
//*******************************************************************//
module Decode(dcA,dcP,   A,NBS,en,rNp);

`define Decode_delay 40

output [30:0] dcA;
reg [30:0] dcA;
output [4:0] dcP;
reg [4:0] dcP;
input [6:0] NBS;
input [3:0] A;
input en,rNp;
reg [30:0] aa;
reg [4:0] pp;
```

```
function [4:0] regno;
input [3:0] nv; input [6:0] NBS;
reg [4:0] rg;
     begin
          if (nv==15)
               rg = 30;
          else
               begin
                    rg = nv;
                    case (NBS)
                         7'h6e :   ;
                         7'h7d :   if (nv>7)
                                        rg = (nv+7);
                         7'h7a :   if (nv > 12)
                                        rg = (nv+9);
                         7'h76 :   if (nv > 12)
                                        rg = (nv+11);
                         7'h5e :   if (nv > 12)
                                        rg = (nv+13);
                         7'h3e :   if (nv > 12)
                                        rg = (nv+15);
                         default : $display("Illegal Bank Select
                                             in Reg Decoder!! NBS=%d, nv=%d", NBS, nv);
                    endcase
               end
          regno = rg;
     end
endfunction

function [2:0] spsno;
input [6:0] NBS;
reg [2:0] sps;
     begin
          sps = 0;
          case (NBS)
               7'h7d :   ;
               7'h7a :   sps = 1;
               7'h76 :   sps = 2;
               7'h5e :   sps = 3;
               7'h3e :   sps = 4;
               default : $display("Illegal SPSR Select in Reg Decoder!! NBS=%d", NBS);
          endcase
          spsno = sps;
     end
endfunction

always @(A or NBS or en or rNp)
     begin
          aa = 0; pp = 0;
          if (en)
               if (rNp)
                    aa = 31'b1<<regno(A,NBS);
               else
                    pp = 5'b1<<spsno(NBS);
          #(`Decode_delay) dcA = aa;
          dcP = pp;
     end

endmodule // Decode
```

```
//*********************************************************************//
//                        Data Input Block                             //
//*********************************************************************//
module Din_blk (out, in, byte, bw);

`define Din_delay 73

output [31:0] out;
reg [31:0] out;
input [31:0] in;
input [1:0] byte;
input bw;
reg [31:0] val, sh;

always @(in or byte or bw)
     begin
          sh = byte * 8;

          val = (sh>0) ? ((in>>sh) | (in<<(32-sh))) : in;

          #(`Din_delay) out = bw ? ((in>>sh) & 'hFF) : val;
     end

endmodule


//*********************************************************************//
//                        Data Output Block                            //
//*********************************************************************//
module Dout_blk (out, in, bw);

`define Dout_delay 35

output [31:0] out;
reg [31:0] out;
input [31:0] in;
input bw;

always @(in or bw)
     begin
          #(`Dout_delay) out = bw ? ((in & 'hFF) * 'h01010101) : in;
     end

endmodule


//*********************************************************************//
//                   Immediate Field Extractor Block                   //
//*********************************************************************//
module ImmExt (out, in);

`define Imm_delay 73

output [31:0] out;
reg [31:0] out;
input [31:0] in;
reg [31:0] val;
```

```
always @(in)
     case((in>>26)&3)
                                                       // LDR/STR 12-bit offset
          2'b01:    #(‘Imm_delay) out = (in & ‘hFFF);
                                                       // Sign-extend branch offset
          2'b10:    begin
                         val = (in & ‘hFFFFFF);
                         if (in & ‘h800000)
                              val = val | ‘hFF000000;
                         #(‘Imm_delay) out = val;
                    end
                                                       // 8-bit immediate
          default:  #(‘Imm_delay) out = (in & ‘hFF);

     endcase
endmodule


//*********************************************************************//
//                         Address Incrementer                         //
//*********************************************************************//
module Incre(out[31:0], in[31:0], rin, rout, aout, rst);

‘define IncDin_Dout 27
‘define IncRst_Rout 47
‘define IncRin_Rout 400
‘define IncAout_Rout 750

output [31:0] out;
reg [31:0] out;
output rout;
reg rout, ack, req;
input [31:0] in;
input rin, aout, rst;

‘define Inc_reset begin ack=1; req=0; out=in; #(‘IncRst_Rout) rout=0; end
‘define Inc_undef begin ack=1; req=0; out='bx; rout='bx; end

always @(rst)
     case(rst)
          1'b1:     ‘Inc_reset
          1'bx:     ‘Inc_undef
     endcase

always @(in)
     if (!req)
          #(‘IncDin_Dout) out = in;

always @(rin)
     begin
          if (rin === 1'bx)
               ‘Inc_undef
          else if (!rst)
               req = 1;
          if (ack & req & !rst)
               begin
                    ack = 0;
                    req = 0;
                    #(‘IncRin_Rout) rout = ~rout;
                    if (in === 32'hFFFFFFFC)
                         out = 0;
                    else
                         out = in + 4;
               end
     end

always @(aout)
     begin
          if (aout === 1'bx)
               ‘Inc_undef
          else if (!rst)
               ack = 1;
          if (ack && req && !rst)
               begin
                    ack = 0;
                    req = 0;
                    #(‘IncAout_Rout) rout = ~rout;
                    if (in === 32'hFFFFFFFC)
                         out = 0;
                    else
                         out = in + 4;
               end
     end

endmodule


//*********************************************************************//
//                              NGen block                             //
//*********************************************************************//
module NGenX(rout, out[3:0],       rin, in[15:0], rst);

// NOTE: Implement using Carry-Save adders ??

‘define NGenRst_Rout 50
‘define NGenRin_Rout 100

output [3:0] out;
reg [3:0] out;
output rout;
reg rout;
input [15:0] in;
input rin, rst;
reg [4:0] cnt, index;

‘define NGen_reset begin out='bx; #(‘NGenRst_Rout) rout=0; end
‘define NGen_undef begin out='bx; rout='bx; end

always @(rst)
     case(rst)
          1'b1:     ‘NGen_reset
          1'bx:     ‘NGen_undef
     endcase

always @(rin)
     begin
          if (rin === 1'bx)
               ‘NGen_undef
```

```
        else if (!rst)
            begin
                cnt = 0;
                for (index = 0; index < 16; index = index +1)
                    if ((in >> index) & 'b1)
                        cnt = cnt + 1;

                cnt = (cnt - 1) & 'hF;

                #('NGenRin_Rout) out = cnt;
                #20 rout = !rout;
            end
    end

endmodule


//**********************************************************************//
//                         RdGen block                                 //
//**********************************************************************//
module RdGen(ain,rout,out[3:0],nTRM,r15,     rin,aout,in[15:0],Rst);

'define RdGenRst_Rout 50
'define RdGenDout 500

output [3:0] out;
reg [3:0] out;
output ain,rout,nTRM,r15;
reg ain,rout,nTRM,r15;
input [15:0] in;
input rin,aout,Rst;

reg [15:0] mask;
reg [4:0] i;
reg [3:0] p;
reg Oack,done;

always @(Rst)
    case(Rst)
        1'b1:    begin
                    Oack = 1;
                    done = 1;
                    #'RdGenRst_Rout rout = 0;
                    ain = 0;
                    nTRM = 'bx;
                    r15 = 'bx;
                    out = 4'hF;
                end
        1'bx:    begin
                    Oack = 1;
                    done = 1;
                    rout = 'bx;
                    ain = 'bx;
                    nTRM = 'bx;
                    r15 = 'bx;
                    out = 'bx;
                end
    endcase

always @(rin)
    if (!Rst)
        begin
            done = 0;
            mask = 16'hFFFF;

            if (Oack && !done)
                begin
                    p = 15;
                    for (i=16; i>0; i=i-1)
                        if (((mask & in) & (1<<(i-1))) >> (i-1))
                            p = (i-1);
                    Oack = 0;
                    mask = (mask & (~(1<<p)));
                    done = ((mask & in) == 0);
                    #'RdGenDout out = p;
                    nTRM = !done;
                    r15 = (p == 4'hF);
                    #50 rout = !rout;
                end
        end

always @(aout)
    if (!Rst)
        begin
            Oack = 1;
            nTRM = 'bx;
            r15 = 'bx;
            if (done)
                begin
                    out = 4'hF;
                    ain = !ain;
                end
            else
            out = 'bx;

            if (Oack && !done)
                begin
                    p = 15;
                    for (i=16; i>0; i=i-1)
                        if (((mask & in) & (1<<(i-1))) >> (i-1))
                            p = (i-1);
                    Oack = 0;
                    mask = (mask & (~(1<<p)));
                    done = ((mask & in) == 0);
                    #'RdGenDout out = p;
                    nTRM = !done;
                    r15 = (p == 4'hF);
                    #50 rout = !rout;
                end
        end

endmodule
```

```
//**************************************************************************************//
//                                                                                      //
//                    The following file contains the behavioural models               //
//                         of the Micropipeline event control blocks                   //
//                                                                                      //
//**************************************************************************************//
`timescale 10ps /10ps

//*********************************************************************//
//                       Event Select Element                          //
//*********************************************************************//
module Select(false, true, in, sel, rst);

`define in_delay 61
`define rst_delay 47

output false, true;
reg false, true;
input in, sel, rst;

initial
     if (rst)
          begin
               #(`rst_delay) true = 0;
               false = 0;
          end

always @(rst)
     case(rst)
          1'b1:     begin
                         #(`rst_delay) true = 0;
                         false = 0;
                    end
          1'bx:     begin
                         #(`rst_delay) true = `bx;
                         false = `bx;
                    end
     endcase

always @(in)
     if (in === 1'bx)
          begin
               #(`in_delay) true = `bx;
               false = `bx;
          end
     else
          if (!rst)
               begin
                    if (sel)
                         #(`in_delay) true = ~true;
                    else if (!sel)
                         #(`in_delay) false = ~false;
                    else
                         begin
                              #(`in_delay) true = `bx;
                              false = `bx;
                         end
               end
endmodule

//*********************************************************************//
//                       4-way Event Select Element                    //
//*********************************************************************//
module Select4(s00, s01, s10, s11, in, bool, rst);

output s00, s01, s10, s11;
input [1:0] bool;
input in, rst;

wire f1, t1;

Select    s1   (f1, t1,       in, bool[1], rst);
Select    s2   (s00, s01,     f1, bool[0], rst);
Select    s3   (s10, s11,     t1, bool[0], rst);

endmodule


//*********************************************************************//
//                       Event Toggle Element                          //
//*********************************************************************//
module Toggle(dot, blank, in, rst);

`define Tog_in_delay 49
`define Tog_rst_delay 47

output dot, blank;
reg dot, blank;
input in, rst;

reg dot_next;

initial
     if (rst)
          begin
               dot_next = 1;
               #(`Tog_rst_delay) dot = 0;
               blank = 0;
          end

always @(rst)
     case(rst)
          1'b1:     begin
                         dot_next = 1;
                         #(`Tog_rst_delay) dot = 0;
                         blank = 0;
                    end
          1'bx:     begin
                         #(`Tog_rst_delay) dot = `bx;
                         blank = `bx;
                    end
     endcase

always @(in)
     if (!rst)
          case(in)
               1'bx:     begin
                              #(`Tog_in_delay) dot = `bx;
                              blank = `bx;
```

```
                    end
            default:  if (dot_next)
                          begin
                              dot_next = 0;
                              #(`Tog_in_delay) dot = ~dot;
                          end
                      else
                          begin
                              dot_next = 1;
                              #(`Tog_in_delay) blank = ~blank;
                          end
        endcase

endmodule


//*********************************************************************//
//                              Call Element                            //
//*********************************************************************//
module Call(done1, done2, Rout, req1, req2, Aout, rst);

`define req1_Rout 27
`define req2_Rout 42
`define Aout_done1 85
`define Aout_done2 85
`define rst_done1 29
`define rst_done2 29

output done1, done2, Rout;
reg done1, done2, Rout;
input req1, req2, Aout, rst;

reg [1:0] caller;

`define call_reset  fork
                         Rout = (req1 ^ req2);
                         caller=0;
                         #(`rst_done1)done1=0;
                         #(`rst_done2)done2=0;
                    join
`define call_undef fork Rout = (req1 ^ req2);caller=0;done1='bx;done2='bx;join


always @(rst)
     case(rst)
          1'b0:     Rout = (req1 ^ req2);
          1'b1:     `call_reset
          1'bx:     `call_undef
     endcase

always @(req1)
     begin
          if (req1 === 1'bx)
               `call_undef
          else if (!rst)
               begin
                    if (caller == 2)
                         begin
                              $display("%m: Call request clash!!");
                              $display("Time=%t", $time);
                         end
                    caller = 1;
               end
          #(`req1_Rout) Rout = (req1 ^ req2);
     end

always @(req2)
     begin
          if (req2 === 1'bx)
               `call_undef
          else if (!rst)
               begin
                    if (caller == 1)
                         begin
                              $display("%m: Call request clash!!");
                              $display("Time=%t", $time);
                         end
                    caller = 2;
               end
          #(`req2_Rout) Rout = (req1 ^ req2);
     end

always @(Aout)
     if (Aout === 1'bx)
          `call_undef
     else if (!rst)
          begin
               case(caller)
                    0:   begin
                              $display("%m: Call acknowledge clash!!");
                              $display("Time=%t", $time);
                         end
                    1:   #(`Aout_done1) done1 = ~done1;
                    2:   #(`Aout_done2) done2 = ~done2;
               endcase
               caller = 0;
          end

endmodule


//*********************************************************************//
//                              4-way Call Element                      //
//*********************************************************************//
module Call4(r1, r2, r3, r4, d1, d2, d3, d4, Rout, Aout, rst);

output d1, d2, d3, d4, Rout;
input r1, r2, r3, r4, Aout, rst;

wire Req1, Req2, Done1, Done2;

Call c1   (d1, d2, Req1, r1, r2, Done1, rst);
Call c2   (d3, d4, Req2, r3, r4, Done2, rst);
Call c3   (Done1, Done2, Rout, Req1, Req2, Aout, rst);

endmodule
```

```
//*********************************************************************//
//                  2-input Muller - C (with Reset)                   //
//*********************************************************************//
module Cgate2(out, a, b, rst);

`define Cgate2A_delay 47
`define Cgate2B_delay 64
`define Cgate2rst_delay 37

output out;
reg out;
input a, b, rst;

initial
     if (rst)
          #(`Cgate2rst_delay) out = 0;

always @(a)
     if (!rst)
          if ((a===b) || (a==='bx))
               #(`Cgate2A_delay) out = a;

always @(b)
     if (!rst)
          if ((a===b) || (b==='bx))
               #(`Cgate2B_delay) out = b;

always @(rst)
     case(rst)
          1'b1:     #(`Cgate2rst_delay) out = 0;
          1'b0:     if (a === b)
                         #(`Cgate2B_delay) out = b;
          1'bx:     #(`Cgate2rst_delay) out = `bx;
     endcase

endmodule


//*********************************************************************//
//                  3-input Muller-C gate (with Reset)                //
//*********************************************************************//
module Cgate3(out, a, b, c, rst);

`define Cgate3A_delay 64
`define Cgate3B_delay 94
`define Cgate3C_delay 111
`define Cgate3rst_delay 37

output out;
reg out;
input a, b, c, rst;

initial
     if (rst)
          #(`Cgate3rst_delay) out = 0;

always @(a)
     if (!rst)
          if (((a===b) && (a===c)) || (a==='bx))
               #(`Cgate3A_delay) out = a;

always @(b)
     if (!rst)
          if (((b===a) && (b===c)) || (b==='bx))
               #(`Cgate3B_delay) out = b;

always @(c)
     if (!rst)
          if (((c===a) && (c===b)) || (c==='bx))
               #(`Cgate3C_delay) out = c;

always @(rst)
     case(rst)
          1'b1:     #(`Cgate3rst_delay) out = 0;
          1'b0:     if ((a===b) && (a===c))
                         #(`Cgate3C_delay) out = a;
          1'bx:     #(`Cgate3rst_delay) out = `bx;
     endcase

endmodule


//*********************************************************************//
//              Event Arbitration Element (No ID indicator)            //
//*********************************************************************//
module Arbit(grant1, grant2, req1, req2, done1, done2, rst);

input req1, req2, done1, done2, rst;
output grant1, grant2;

wire active;

ArbitA    arb  (grant1, grant2, active, req1, req2, done1, done2, rst);
Sink      x    (active);

endmodule


//*********************************************************************//
//                        Event Arbitration Element                    //
//*********************************************************************//
module ArbitA(grant1, grant2, active, req1, req2, done1, done2, rst);

`define R1_G1 221
`define D2_G1 227
`define rst_G1 50
`define R2_G2 179
`define D1_G2 220
`define rst_G2 50
`define R1_ID 165
`define D1_ID 121
`define D2_ID 170
`define rst_ID 172

output grant1, grant2, active;
reg grant1, grant2, active;
input req1, req2, done1, done2, rst;
reg [1:0] caller;
```

```
`define arbit_reset fork
                        #(`rst_ID)active=0;
                        caller=0;
                        #(`rst_G1)grant1=0;
                        #(`rst_G2)grant2=0;
                    join
`define arbit_undef fork active='bx;caller=0;grant1='bx;grant2='bx;join

always @(rst)
     case(rst)
          1'b1:     `arbit_reset
          1'bx:     `arbit_undef
     endcase

always @(done1)
     if (!rst)
          if (done1 === 1'bx)
               `arbit_undef
          else
               begin
                    if (caller != 1)
                         begin
                              $display("%m: Arbiter done1 clash!!");
                              $display("Time=%t", $time);
                         end
                    else
                         caller = 0;
                    fork
                         if (req2 != grant2)
                              begin
                                   caller = 2;
                                   #(`D1_G2)grant2 = req2;
                              end
                         #(`D1_ID) active = 0;
                    join
               end

always @(done2)
     if (!rst)
          if (done2 === 1'bx)
               `arbit_undef
          else
               begin
                    if (caller != 2)
                         begin
                              $display("%m: Arbiter done2 clash!!");
                              $display("Time=%t", $time);
                         end
                    else
                         caller = 0;
                    if (req1 != grant1)
                         fork
                              caller = 1;
                              #(`D2_ID) active = 1;
                              #(`D2_G1) grant1 = req1;
                         join
               end

always @(req1 or req2)
     if (!rst)
          if ((req1 === 1'bx) || (req2 === 1'bx))
               begin
                    if ((req1 === 1'bx) && (caller != 2))
                         `arbit_undef
                    if ((req2 === 1'bx) && (caller != 1))
                         `arbit_undef
               end
          else
               begin
                    if ((req2 != grant2) && (caller == 0))
                         fork
                              caller = 2;
                              #(`R2_G2) grant2 = req2;
                         join
                    if ((req1 != grant1) && (caller == 0))
                         fork
                              caller = 1;
                              #(`R1_ID) active = 1;
                              #(`R1_G1) grant1 = req1;
                         join
               end

endmodule


//*********************************************************************//
//                        Capture-Pass Register                        //
//*********************************************************************//
module CP_Reg(out, in, capture, pass);
parameter width = 1;

`define data_delay 32
`define pass_delay 63

output [width-1:0] out;
reg [width-1:0] out;
input [width-1:0] in;
input capture, pass;

always @(in)
     if (capture == pass)
          #(data_delay) out = in;

always @(capture or pass)
     if (capture == pass)
          #(pass_delay) out = in;
endmodule


//*********************************************************************//
//                         2 by 1 Decision Wait                        //
//*********************************************************************//
module DecWait2x1(out1, out2, req, c1, c2, rst);

`define DWrst_delay 29
`define DWreq_delay 85
`define DWcx_delay 47
```

```
output out1, out2;
reg out1, out2;
input req, c1, c2, rst;

reg req_in, c1_in, c2_in, block;

`define DW2x1_reset fork
                        req_in = 0;
                        c1_in = 0;
                        c2_in = 0;
                        block = 0;
                        #(`DWrst_delay) out1 = 0;
                        out2 = 0;
                   join
`define DW2x1_undef begin req_in = 0;c1_in = 0;c2_in = 0;block = 0;out1 = `bx;out2 = `bx; end

initial
     if (rst)
          `DW2x1_reset

always @(rst)
     case(rst)
          1'b1:     `DW2x1_reset
          1'bx:     `DW2x1_undef
     endcase

always @(req)
     if (req === 1'bx)
          `DW2x1_undef
     else if (!rst)
          begin
               if (req_in)
                    begin
                         $display("%m: DW2x1 request clash!!",);
                         $display("Time=%t", $time);
                    end
               req_in = 1;
               if (req_in && c1_in && c2_in)
                    block = 1;
               fork
                    if (req_in && c1_in)
                         begin
                              req_in = 0;
                              c1_in = 0;
                              #(`DWreq_delay) out1 = ~out1;
                         end
                    if (req_in && c2_in && !block)
                         begin
                              req_in = 0;
                              c2_in = 0;
                              #(`DWreq_delay) out2 = ~out2;
                         end
               join
               block = 0;
          end

always @(c1)
     if (c1 === 1'bx)
          `DW2x1_undef
     else if (!rst)
          begin
               if (c1_in)
                    begin
                         $display("%m: DW2x1 c1 clash!!",);
                         $display("Time=%t", $time);
                    end
               c1_in = 1;
               if (req_in && c1_in)
                    begin
                         req_in = 0;
                         c1_in = 0;
                         #(`DWcx_delay) out1 = ~out1;
                    end
          end

always @(c2)
     if (c2 === 1'bx)
          `DW2x1_undef
     else if (!rst)
          begin
               if (c2_in)
                    begin
                         $display("%m: DW2x1 c2 clash!!",);
                         $display("Time=%t", $time);
                    end
               c2_in = 1;
               if (req_in && c2_in)
                    begin
                         req_in = 0;
                         c2_in = 0;
                         #(`DWcx_delay) out2 = ~out2;
                    end
          end

endmodule


//*********************************************************************//
//                         Event (driven) Register                      //
//*********************************************************************//
module EvtReg(out, Ain, Rout, in, Rin, Aout, rst);
parameter width = 1;

`define EregRst_delay 110
`define EregRin_Ain 120
`define EregRin_Rout 120
`define EregAout_Ain 220
`define EregAout_Rout 220
`define EregAout_Dout 111
`define EregDin_Dout 32

output [width-1:0] out;
reg [width-1:0] out;
output Ain, Rout;
reg Ain, Rout;
input [width-1:0] in;
input Rin, Aout, rst;
```

```
reg pass, inRdy;
time DataT, ReqT, BundT, MinT;

`define Ereg_reset fork pass=1; inRdy=0; #(`EregRst_delay) out=in; Ain=0; Rout=0; join
`define Ereg_undef begin pass=1; inRdy=0; out='bx; Ain='bx; Rout='bx; end

initial
     begin
          ReqT = 0; DataT = 0; BundT = 0;
          MinT = 99999999;                        // some arbitrary large value !!
     end

always @(rst)
     case(rst)
          1'b0:     out = in;
          1'b1:     begin
                         if (($time > 10000) && (BundT != 0))
                              $display("EvtReg %m: %0d @ %0d", (MinT*10), BundT);
                         `Ereg_reset
                    end
          1'bx:     `Ereg_undef
     endcase

always @(in)
     begin
          if (!rst)
               DataT = $time;
          if (inRdy)
               $display ("Time=%0t - EvtReg %m: Data change after Req", $time);
          if (pass)
               #(`EregDin_Dout) out = in;
     end

always @(Rin)
     if (!rst)
          begin
               ReqT = $time;
               if ((ReqT - DataT) < MinT)
                    begin
                         MinT = (ReqT - DataT);
                         BundT = ReqT;
                    end
               if (Rin === 1'bx)
                    `Ereg_undef
               inRdy = 1;
               if (pass && inRdy)
                    fork
                         #(`EregDin_Dout) out = in;
                         #(`EregRin_Rout) Rout = ~Rout;
                         #(`EregRin_Ain) Ain = ~Ain;
                         pass = 0;
                         inRdy = 0;
                    join
          end

always @(Aout)
     if (!rst)
          begin
               if (Aout === 1'bx)
                    `Ereg_undef
               pass = 1;
               if (pass && inRdy)
                    fork
                         #(`EregAout_Dout) out = in;
                         #(`EregAout_Rout) Rout = ~Rout;
                         #(`EregAout_Ain) Ain = ~Ain;
                         pass = 0;
                         inRdy = 0;
                    join
               else
                    #(`EregAout_Dout) out = in;
          end

endmodule


//*********************************************************************//
//                  Event (driven) Register (2 bundles)               //
//*********************************************************************//
module EvtReg2(out1, out2, Ain, Rout, in1, in2, Rin, Aout, rst);
parameter width1 = 1, width2 = 1;

`define Rst_delay 110
`define Rin_Ain 120
`define Rin_Rout 120
`define Aout_Ain 220
`define Aout_Rout 220
`define Aout_Dout 111
`define Din_Dout 32

output [width1-1:0] out1;
reg [width1-1:0] out1;
output [width2-1:0] out2;
reg [width2-1:0] out2;
output Ain, Rout;
reg Ain, Rout;
input [width1-1:0] in1;
input [width2-1:0] in2;
input Rin, Aout, rst;

reg pass, inRdy;
time DataT, ReqT, BundT, MinT;

`define Ereg2_reset fork pass=1; inRdy=0; #(`Rst_delay) out1=in1; out2=in2; Ain=0; Rout=0; join
`define Ereg2_undef begin pass=1; inRdy=0; out1='bx; out2='bx; Ain='bx; Rout='bx; end

initial
     begin
          ReqT = 0; DataT = 0; BundT = 0;
          MinT = 99999999;                        // some arbitrary large value !!
     end

always @(rst)
     case(rst)
          1'b0:     begin out1 = in1; out2 = in2; end
          1'b1:     begin
                         if (($time > 10000) && (BundT != 0))
                              $display("EvtReg2 %m: %0d @ %0d", (MinT*10), BundT);
```

```
                    `Ereg2_reset
                end
            1'bx:   `Ereg2_undef
        endcase

always @(in1 or in2)
    begin
        if (!rst)
            DataT = $time;
        if (inRdy)
            $display ("Time=%0t - EvtReg2 %m: Data change after Req", $time);
        if (pass)
            begin
                #(`Din_Dout) out1 = in1;
                out2 = in2;
            end
    end

always @(Rin)
    if (!rst)
        begin
            ReqT = $time;
            if ((ReqT - DataT) < MinT)
                begin
                    MinT = (ReqT - DataT);
                    BundT = ReqT;
                end
            if (Rin === 1'bx)
                `Ereg2_undef
            inRdy = 1;
            if (pass && inRdy)
                fork
                    #(`Din_Dout) out1 = in1;
                    #(`Din_Dout) out2 = in2;
                    #(`Rin_Rout) Rout = ~Rout;
                    #(`Rin_Ain) Ain = ~Ain;
                    pass = 0;
                    inRdy = 0;
                join
        end

always @(Aout)
    if (!rst)
        begin
            if (Aout === 1'bx)
                `Ereg2_undef
            pass = 1;
            if (pass && inRdy)
                fork
                    #(`Aout_Dout) out1 = in1;
                    #(`Aout_Dout) out2 = in2;
                    #(`Aout_Rout) Rout = ~Rout;
                    #(`Aout_Ain) Ain = ~Ain;
                    pass = 0;
                    inRdy = 0;
                join
            else
                begin
                    #(`Aout_Dout) out1 = in1;
                    out2 = in2;
                end
        end

endmodule


//*********************************************************************//
//                          Event (driven) Latch                        //
//*********************************************************************//
module EvtLch(out, in, Rin, rst);
parameter width = 1;

`define ElchRst_delay 74
`define ElchRin_Out 43

output [width-1:0] out;
reg [width-1:0] out;
input [width-1:0] in;
input Rin, rst;

time DataT, ReqT, BundT, MinT;

initial
    begin
        ReqT = 0; DataT = 0; BundT = 0;
        MinT = 99999999;                        // some arbitrary large value !!
    end

always @(rst)
    case(rst)
        1'b1:   begin
                    if (($time > 10000) && (BundT != 0))
                        $display("EvtLch %m: %0d @ %0d", (MinT*10), BundT);
                    #(`ElchRst_delay) out=0;
                end
        1'bx:   out='bx;
    endcase

always @(in)
    if (!rst)
        DataT = $time;

always @(Rin)
    if (!rst)
        begin
            ReqT = $time;
            if ((ReqT - DataT) < MinT)
                begin
                    MinT = (ReqT - DataT);
                    BundT = ReqT;
                end
            if (Rin === 1'bx)
                out='bx;
            else
                #(`ElchRin_Out) out = in;
        end

endmodule
```

```
//**********************************************************************//
//                        Lock FIFO Register                            //
//**********************************************************************//
module LkReg(out,outP,psO,lkO,lPO,Ain,Rout,
                         in,inP,psI,lkI,lPI,Rin,Aout,rst);

`define LregRst_delay 110
`define LregRin_Ain 120
`define LregRin_Rout 120
`define LregAout_Ain 220
`define LregAout_Rout 220
`define LregAout_Dout 111
`define LregDin_Dout 32
`define LregOrA_delay 26
`define LregOrB_delay 39

output [29:0] out,lkO;
reg [29:0] out,lkO;
output [4:0] outP,lPO;
reg [4:0] outP,lPO;
output [1:0] psO;
reg [1:0] psO;
output Ain, Rout;
reg Ain, Rout;
input [29:0] in,lkI;
input [4:0] inP,lPI;
input [1:0] psI;
input Rin, Aout, rst;

reg pass, inRdy;

`define Lreg_reset begin pass=1; inRdy=0; #(`LregRst_delay) Ain=0; Rout=0; end
`define Lreg_undef begin pass=1; inRdy=0; Ain='bx; Rout='bx; end

always @(rst)
     case(rst)
          1'b0:     begin out = in; outP = inP; psO = psI; end
          1'b1:     `Lreg_reset
          1'bx:     `Lreg_undef
     endcase

always @(psI)
     if (pass)
          #(`LregDin_Dout) psO = psI;

always @(in)
     if (pass)
          begin
               #(`LregDin_Dout) out = in;
               #(`LregOrB_delay) lkO = (out | lkI);
          end

always @(inP)
     if (pass)
          begin
               #(`LregDin_Dout) outP = inP;
               #(`LregOrB_delay) lPO = (outP | lPI);
          end

always @(lkI)
     #(`LregOrA_delay) lkO = (out | lkI);

always @(lPI)
     #(`LregOrA_delay) lPO = (outP | lPI);

always @(Rin)
     if (!rst)
          begin
               if (Rin === 1'bx)
                    `Lreg_undef
               inRdy = 1;
               if (pass && inRdy)
                    fork
                         begin
                              #(`LregDin_Dout) out = in;
                              #(`LregOrB_delay) lkO = (out | lkI);
                         end
                         begin
                              #(`LregDin_Dout) outP = inP;
                              #(`LregOrB_delay) lPO = (outP | lPI);
                         end
                         #(`LregDin_Dout) psO = psI;
                         #(`LregRin_Rout) Rout = ~Rout;
                         #(`LregRin_Ain) Ain = ~Ain;
                         pass = 0;
                         inRdy = 0;
                    join
          end

always @(Aout)
     if (!rst)
          begin
               if (Aout === 1'bx)
                    `Lreg_undef
               pass = 1;
               if (pass && inRdy)
                    fork
                         begin
                              #(`LregAout_Dout) out = in;
                              #(`LregOrB_delay) lkO = (out | lkI);
                         end
                         begin
                              #(`LregAout_Dout) outP = inP;
                              #(`LregOrB_delay) lPO = (outP | lPI);
                         end
                         #(`LregAout_Dout) psO = psI;
                         #(`LregAout_Rout) Rout = ~Rout;
                         #(`LregAout_Ain) Ain = ~Ain;
                         pass = 0;
                         inRdy = 0;
                    join
               else
                    fork
                         begin
                              #(`LregAout_Dout) out = in;
                              #(`LregOrB_delay) lkO = (out | lkI);
                         end
                         begin
```

```
                              #(`LregAout_Dout) outP = inP;
                              #(`LregOrB_delay) lPO = (outP | lPI);
                         end
                         #(`LregAout_Dout) psO = psI;
                    join
          end

endmodule


//**********************************************************************//
//                       2-stage FIFO Pipeline
//**********************************************************************//
module Pipe2(out, Ain, Rout, in, Rin, Aout, rst);

output [31:0] out;
output Ain, Rout;
input [31:0] in;
input Rin, Aout, rst;

wire [31:0] out1;
wire A1, R1;

EvtReg #32     e1   (out1, Ain, R1,     in, Rin, A1, rst);
EvtReg #32     e2   (out, A1, Rout,     out1, R1, Aout, rst);

endmodule


//**********************************************************************//
//                       3-stage FIFO Pipeline
//**********************************************************************//
module Pipe3(out, Ain, Rout, in, Rin, Aout, rst);

output [31:0] out;
output Ain, Rout;
input [31:0] in;
input Rin, Aout, rst;

wire [31:0] out1, out2;
wire A1, A2, R1, R2;

EvtReg #32     e1   (out1, Ain, R1,     in, Rin, A1, rst);
EvtReg #32     e2   (out2, A1, R2,      out1, R1, A2, rst);
EvtReg #32     e3   (out, A2, Rout,     out2, R2, Aout, rst);

endmodule
```

```
//*************************************************************************************//
//                                                                                     //
//             The following file contains the standard gate functions library         //
//                    (customised to reflect the different propagation                 //
//                      delays of the various MDCML switching levels)                  //
//                                                                                     //
//*************************************************************************************//

// Contents of the miscellaneous functions library.
`timescale 10ps /10ps

//              DLinv(out, in);
//              DLbuff(out, in);
//              PwrDrv(out, in);
//              DLand2(out, a, b);
//              DLor2(out, a, b);
//              DLxor2(out, a, b);
//              DLand3(out, a, b, c);
//              DLor3(out, a, b, c);
//              DLxor3(out, a, b, c);
//              DLand4(out, a, b, c, d);
//              DLor4(out, a, b, c, d);
//#(width)      Mux2(out, a, b, sel);
//#(width)      Mux4(out, a, b, c, d, sel[1:0]);
//#(width)      LatchR(out, in, enable, rst);
//#(width)      Latch(out, in, enable);
//#(width)      Band(out, a, b);
//#(width)      Binv(out, in);
//#(width)      Bbuff(out, in);
//#(width)      BNbuff(out, in);
//#(width)      TriDrv(out, in, enb);
//#(width)      Periph(out, in);
//#(width)      Sink(in);
//              MemTriDrv(out, in, req, ack);

//*********************************************************************//
//                  Differential logic "INVERTER"                      //
//*********************************************************************//
module DLinv(out, in);
output out;
input in;

assign out = ~in;

endmodule


//*********************************************************************//
//                  Differential logic BUFFER gate                     //
//*********************************************************************//
module DLbuff(out, in);
parameter buff_delay = 25;

output out;
input in;

assign #buff_delay out = in;

endmodule
```

```
//*********************************************************************//
//                  Differential logic Power Driver                    //
//*********************************************************************//
module PwrDrv(out, in);

`define PwrDrv_delay 60

output out;
input in;

assign #`PwrDrv_delay out = in;

endmodule


//*********************************************************************//
//                  Differential logic 2-input AND gate                //
//*********************************************************************//
module DLand2(out, a, b);

`define and2A_delay 26
`define and2B_delay 42

output out;
input a, b;
wire delb;

assign #(`and2B_delay-`and2A_delay)     delb = b;
assign #(`and2A_delay)                  out = (a & delb);

endmodule


//*********************************************************************//
//                  Differential logic 2-input OR gate                 //
//*********************************************************************//
module DLor2(out, a, b);

`define or2A_delay 26
`define or2B_delay 39

output out;
input a, b;
wire delb;

assign #(`or2B_delay-`or2A_delay)  delb = b;
assign #(`or2A_delay)              out = (a | delb);

endmodule


//*********************************************************************//
//                  Differential logic 2-input XOR gate                //
//*********************************************************************//
module DLxor2(out, a, b);

`define xor2A_delay 27
`define xor2B_delay 42
```

```
output out;
input a, b;
wire delb;

assign #(`xor2B_delay-`xor2A_delay)     delb = b;
assign #(`xor2A_delay)                  out = (a ^ delb);

endmodule


//*********************************************************************//
//                  Differential logic 3-input AND gate                //
//*********************************************************************//
module DLand3(out, a, b, c);

`define and3A_delay 28
`define and3B_delay 40
`define and3C_delay 52

output out;
input a, b, c;
wire delb, delc;

assign #(`and3C_delay-`and3A_delay)     delc = c;
assign #(`and3B_delay-`and3A_delay)     delb = b;
assign #(`and3A_delay)                  out = (a & delb & delc);

endmodule


//*********************************************************************//
//                  Differential logic 3-input OR gate                 //
//*********************************************************************//
module DLor3(out, a, b, c);

`define or3A_delay 27
`define or3B_delay 44
`define or3C_delay 56

output out;
input a, b, c;
wire delb, delc;

assign #(`or3C_delay-`or3A_delay)       delc = c;
assign #(`or3B_delay-`or3A_delay)       delb = b;
assign #(`or3A_delay)                   out = (a | delb | delc);

endmodule


//*********************************************************************//
//                  Differential logic 3-input XOR gate                //
//*********************************************************************//
module DLxor3(out, a, b, c);

`define xor3A_delay 26
`define xor3B_delay 44
`define xor3C_delay 59

output out;
input a, b, c;
wire delb, delc;

assign #(`xor3C_delay-`xor3A_delay)     delc = c;
assign #(`xor3B_delay-`xor3A_delay)     delb = b;
assign #(`xor3A_delay)                  out = (a ^ delb ^ delc);

endmodule


//*********************************************************************//
//                  Differential logic 4-input AND gate                //
//*********************************************************************//
module DLand4(out, a, b, c, d);

`define and4A_delay 29
`define and4B_delay 48
`define and4C_delay 58
`define and4D_delay 70

output out;
input a, b, c, d;
wire delb, delc, deld;

assign #(`and4D_delay-`and4A_delay)     deld = d;
assign #(`and4C_delay-`and4A_delay)     delc = c;
assign #(`and4B_delay-`and4A_delay)     delb = b;
assign #(`and4A_delay)                  out = (a & delb & delc & deld);

endmodule


//*********************************************************************//
//                  Differential logic 4-input OR gate                 //
//*********************************************************************//
module DLor4(out, a, b, c, d);

`define or4A_delay 29
`define or4B_delay 48
`define or4C_delay 58
`define or4D_delay 70

output out;
input a, b, c, d;
wire delb, delc, deld;

assign #(`or4D_delay-`or4A_delay)       deld = d;
assign #(`or4C_delay-`or4A_delay)       delc = c;
assign #(`or4B_delay-`or4A_delay)       delb = b;
assign #(`or4A_delay)                   out = (a | delb | delc | deld);

endmodule


//*********************************************************************//
//                       2:1 Multiplexor (any width)                   //
//*********************************************************************//
module mux2(out, a, b, sel);
```

```
parameter width = 1;

`define mux2data_delay 27
`define mux2sel_delay 43

output [width-1:0] out;
reg [width-1:0] out;
input [width-1:0] a, b;
input sel;

initial
     case(sel)
          1'b0:     out = a;
          1'b1:     out = b;
          default:  out = `bx;
     endcase

always @(a)
     if(!sel)
          #(`mux2data_delay) out = a;

always @(b)
     if(sel)
          #(`mux2data_delay) out = b;

always @(sel)
     case(sel)
          1'b0:     #(`mux2sel_delay) out = a;
          1'b1:     #(`mux2sel_delay) out = b;
          default:  out = `bx;
     endcase
endmodule

//*********************************************************************//
//                       4:1 Multiplexor (any width)                   //
//*********************************************************************//
module mux4(out, a, b, c, d, sel);
parameter width = 1;

`define mux4data_delay 33
`define mux4sel1_delay 49
`define mux4sel0_delay 59

output [width-1:0] out;
reg [width-1:0] out;
input [width-1:0] a, b, c, d;
input [1:0] sel;

initial
     case(sel)
          2'b00:    out = a;
          2'b01:    out = b;
          2'b10:    out = c;
          2'b11:    out = d;
          default:  out = `bx;
     endcase

always @(a)
     if(sel === 2'b00)
          #(`mux4data_delay) out = a;

always @(b)
     if(sel === 2'b01)
          #(`mux4data_delay) out = b;

always @(c)
     if(sel === 2'b10)
          #(`mux4data_delay) out = c;

always @(d)
     if(sel === 2'b11)
          #(`mux4data_delay) out = d;

always @(sel[1])
     case(sel)
          2'b00:    #(`mux4sel1_delay) out = a;
          2'b01:    #(`mux4sel1_delay) out = b;
          2'b10:    #(`mux4sel1_delay) out = c;
          2'b11:    #(`mux4sel1_delay) out = d;
          default:  out = `bx;
     endcase

always @(sel[0])
     case(sel)
          2'b00:    #(`mux4sel0_delay) out = a;
          2'b01:    #(`mux4sel0_delay) out = b;
          2'b10:    #(`mux4sel0_delay) out = c;
          2'b11:    #(`mux4sel0_delay) out = d;
          default:  out = `bx;
     endcase

endmodule

//*********************************************************************//
//                Transparent Latch with Reset (any width)             //
//*********************************************************************//
module LatchR(out, in, enable, rst);
parameter width = 1;

`define LatchRrst_delay 47
`define LatchRdata_delay 33
`define LatchRenb_delay 49

output [width-1:0] out;
reg [width-1:0] out;
input [width-1:0] in;
input enable, rst;

initial
     if (rst)
          #(`LatchRrst_delay) out = 0;
     else if (enable)
          out = in;
always @(rst)
     case(rst)
```

```
          1'b0:     if (enable)
                         #(`LatchRenb_delay) out = in;
          1'b1:     #(`LatchRrst_delay) out = 0;
          1'bx:     #(`LatchRrst_delay) out = `bx;
     endcase

always @(enable)
     if (!rst)
          case(enable)
               1'b1:     #(`LatchRenb_delay) out = in;
               1'bx:     out = `bx;
          endcase

always @(in)
     if (!rst && enable)
          #(`LatchRdata_delay) out = in;
endmodule


//*********************************************************************//
//                        Transparent Latch (any width)               //
//*********************************************************************//
module Latch(out, in, enable);
parameter width = 1;

`define Latchdata_delay 33
`define Latchenb_delay 49

output [width-1:0] out;
reg [width-1:0] out;
input [width-1:0] in;
input enable;

initial
     if (enable)
          out = in;

always @(enable)
     case(enable)
          1'b1:     #(`Latchenb_delay) out = in;
          1'bx:     out = `bx;
     endcase

always @(in)
     if (enable)
          #(`Latchdata_delay) out = in;
endmodule


//*********************************************************************//
//                   Bundled 2-input AND gate (any width)             //
//*********************************************************************//
module Band(out, a, b);
parameter width = 1;

`define BandA_delay 26
`define BandB_delay 42
output [width-1:0] out;
reg [width-1:0] out;
input [width-1:0] a, b;

initial
     out = (a & b);

always @(a)
     #(`BandA_delay) out = (a & b);

always @(b)
     #(`BandB_delay) out = (a & b);

endmodule


//*********************************************************************//
//                   Bundled INVERTER gate (any width)                //
//*********************************************************************//
module Binv(out, in);
parameter width = 1;

output [width-1:0] out;
input [width-1:0] in;

assign out = ~in;

endmodule


//*********************************************************************//
//                   Bundled BUFFER gate (any width)                  //
//*********************************************************************//
module Bbuff(out, in);
parameter width = 1;

`define Bbuff_delay 25

output [width-1:0] out;
input [width-1:0] in;

wire [width-1:0] #`Bbuff_delay out = in;

endmodule


//*********************************************************************//
//                   Bundled INVERTING BUFFER (any width)             //
//*********************************************************************//
module BNbuff(out, in);
parameter width = 1;

output [width-1:0] out;
input [width-1:0] in;

wire [width-1:0] out = ~ in;

endmodule
```

```
//**********************************************************************//
//                      TRISTATE BUFFER (any width)                    //
//**********************************************************************//
module TriDrv(out, in, Nenb);
parameter width = 1;

`define TriData_delay 25
`define TriEnb_delay 40
`define TriZ_delay 50

output [width-1:0] out;
reg [width-1:0] out;
input [width-1:0] in;
input Nenb;

always @(in)
     if (!Nenb)
          #(`TriData_delay) out = in;

always @(negedge Nenb)
     if (Nenb==0)
          #(`TriEnb_delay) out = in;
     else
          #(`TriEnb_delay) out = `bz;

always @(posedge Nenb)
     #(`TriZ_delay) out = `bz;

endmodule


//**********************************************************************//
//              BIPOLAR PERIPHERAL BUFFER (any width)                  //
//**********************************************************************//
module Periph(out, in);
parameter width = 1;

`define Peri_delay 150

output [width-1:0] out;
input [width-1:0] in;

wire [width-1:0] #`Peri_delay out = in;

endmodule


//**********************************************************************//
//                                  Sink                               //
//**********************************************************************//
module Sink(in);
parameter width = 1;

input [width-1:0] in;

endmodule
```

```
//**********************************************************************//
//                       Memory Tristate Driver
//**********************************************************************//
module MemTriDrv(out, in, req, ack, rst);

output [31:0] out;
reg [31:0] out;
input [31:0] in;
input req, ack, rst;
reg driving;

initial
     begin
          out = `bz;
          driving = 0;
     end

always @(rst)
     if (rst != 1'b0)
          begin
               out = `bz;
               driving = 0;
          end

always @(in)
     if (driving)
          out = in;

always @(req)
     if (!rst)
          driving = 1;

always @(ack)
     if (!rst)
          begin
               out = `bz;
               driving = 0;
          end

endmodule
```

```
//**************************************************************************************//
//                                                                                      //
//              The following file gives an example of the PLA model structure         //
//                            ( in this case the Mode decode PLA )                      //
//                                                                                      //
//**************************************************************************************//
`timescale 10ps/10ps
// Mode decode PLA model

module Mode_PLA(vect,F, mode);
parameter delay = 100;

output [6:0] mode;
reg [6:0] mode;
input [2:0] vect;
input F;

reg[1:4] and_plane[1:11];
reg[1:11] or_plane[1:7];

reg a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11;
reg o1,o2,o3,o4,o5,o6,o7;

initial
     begin
          $readmemb("/home/r83/users/robertk/Verilog/ARM/pla/Mode.and", and_plane);
          $readmemb("/home/r83/users/robertk/Verilog/ARM/pla/Mode.or", or_plane);

          $async$and$plane(and_plane,   {vect[2],vect[1],vect[0],F},
                                                  {a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11});
          $async$or$plane(or_plane,     {a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11},
                                                  {o1,o2,o3,o4,o5,o6,o7});
     end


always@(o1 or o2 or o3 or o4 or o5 or o6 or o7)
     begin
          #delay mode[6] = o1;
          mode[5] = o2;
          mode[4] = o3;
          mode[3] = o4;
          mode[2] = o5;
          mode[1] = o6;
          mode[0] = o7;
     end
endmodule

// Mode decode PLA model - AND plane
0010
0011
0100
0101
0110
0111
1000
1001
1100
1101
111?
```

```
// Mode decode PLA model - OR plane
11111111111
?1?1?1?1?11
11111111111
11?????????
????1111???
1111111111?
11111111??1
```

```
//***********************************************************************************//
//                                                                                   //
//              The following file contains the external environment model          //
//                           ( MMU and Asynchronous Memory )                         //
//                                                                                   //
//***********************************************************************************//
`timescale 10ps/10ps
//   the async ARM validation harness

module SysST (XBigEnd,XnAbt,XRst);
input XBigEnd,XnAbt,XRst;

tri [31:0] Xd;
wire [31:0] Xa;
wire [1:0] XDabt;

ARMst     cpu       (Xa[31:0],{Seq,Inc,Ren,Wen,Usr,bNw,Opc},XMr,XRa,
                                             Xd[31:0],Xdbe,XNfiq,XNirq,XDabt[1:0],
                                             XPAbt,XMa,XRr,XBigEnd,XnAbt,XRst);

MMUf      mmu       (Mrq1,Pabt0,XDabt[1:0],WrDabt,     Xa[31:16],Inc,Ren,Opc,XMr,XRst);

DLxor2    g0        (XMa,     Ma0,WrDabt);

MemFIQ    mem       (Ma0,XRr,XPAbt,Xdbe,XNirq,XNfiq,
                               Xa[31:0],Xd[31:0],Ren,Wen,bNw,Mrq1,XRa,Pabt0,XRst);

endmodule // System


//   the async ARM chip
module ARMst (Xa[31:0],Xc[6:0],XMr,XRa,
                    Xd[31:0],Xdbe,XNfiq,XNirq,XDabt[1:0],XPAbt,XMa,XRr,XBigEnd,XnAbt,XRst);
`include "ARMst.inc"

ARMstCore core      (Add[31:0],Dout[31:0],Ctl[6:0],Mr,MRRa,
                              MRR[31:0],Nfiq,Nirq,Dabt[1:0],PAbt,Ma,MRRr,BigEnd,nAbt,Rst);

// the data bus IOs
TriDrv #32 Od       (Xd[31:0],     Dout[31:0],Nwe);
Periph #32 Id       (MRR[31:0],    Xd[31:0]);
Periph    ODb       (we,           Xdbe);
DLinv     inv       (Nwe,          we);

// the address outputs
Periph #32 Oa       (Xa[31:0],     Add[31:0]);

// the control outputs
Periph #7 Oc        (Xc[6:0],      Ctl[6:0]);

// requests and acknowledges
Periph    OMr       (XMr,          Mr);
Periph    OMa       (Ma,           XMa);

Periph    ORr       (MRRr,         XRr);
Periph    ORa       (XRa,          MRRa);

// other IOs
Periph    ONf       (Nfiq,         XNfiq);
Periph    ONi       (Nirq,         XNirq);
Periph #2 ODa       (Dabt[1:0],    XDabt[1:0]);
Periph    OPa       (PAbt,         XPAbt);
Periph    OBE       (BigEnd,       BigEnd);
Periph    OnA       (nAbt,         XnAbt);
Periph    OCd       (Rst,          XRst);

endmodule // ARMst


//   Memory access Controller
// Wr.Rq && Perr :- Dabt[1], No Mem Cycle ???? Load Mult ???
// Rd.Rq && OPC && !Perr :- Cycle Mem Normal Pre-Fetch
// Rd.Rq && OPC && Perr :- Cycle Mem, Flag data
// Rd.Rq && !OPC && !INC && !Perr :- Dabt[0], Cycle Mem - Normal activity
// Rd.Rq && !OPC && !INC && Perr :- Dabt[1], Cycle Mem - Flag Data

module MMUf (Mrq1,Pabt,Dabt[1:0],WrDabt, Add[31:16],Inc,Ren,Opc,Mrq0,Rst);

input [31:16] Add;
input Inc,Ren,Opc,Mrq0,Rst;

output [1:0] Dabt;
output Mrq1,Pabt,WrDabt;

// Address bit in range 16-30 ? ==> signal "OUT OF RANGE"
DLor3     sg1       (AddOR1,       Add[30],Add[29],Add[28]);
DLor3     sg2       (AddOR2,       Add[27],Add[26],Add[25]);
DLor3     sg3       (AddOR3,       Add[24],Add[23],Add[22]);
DLor3     sg4       (AddOR4,       Add[21],Add[20],Add[19]);
DLor3     sg5       (AddOR5,       Add[18],Add[17],Add[16]);
DLor3     sg6       (AddOR6,       AddOR1,AddOR2,AddOR3);
DLor3     sg7       (Pabt,         AddOR6,AddOR5,AddOR4);

// Work out necessary response
DLor2     sg8       (nDabtRq,           Opc,Inc);
DLinv     inv       (DabtRq,            nDabtRq);
Select    sel0      (NoDabt,DabtR,      Mrq0,DabtRq,Rst);
Select    sel1      (Dabt[0],Dabt[1],   DabtR,Pabt,Rst);
Select    sel2      (WrDabt,RdDabt,     Dabt[1],Ren,Rst);
DLxor2    xor0      (x0,                NoDabt,Dabt[0]);
DLxor2    xor1      (Mrq1,              x0,RdDabt);

endmodule // MMUf


//   Async Mem I/F
module MemFIQ (MEMak,MRRrq,OPerr,Xdbe,Niq,Fiq,
                              MAR[31:0],MRR[31:0],Ren,Wen,bNw,MEMrq,MRRak,IPerr,Rst);

input [31:0] MAR;
input Ren,Wen,bNw,MEMrq,MRRak,IPerr,Rst;

output MEMak,MRRrq,OPerr,Xdbe,Niq,Fiq;

inout [31:0] MRR;

`define memsize     65536 // bytes
`define memdelay        5 // ns
```

```
DLbuff    bf1       (OPerr,        IPerr);

Select    sl0       (W1,RDrq,      MEMrq,Ren,Rst);
DLxor2    xr0       (WRrq,         W1,W2);
DLxor2    xr1       (Xdbe,         WRrq,WRdn);
Select    sl1       (Rend,W2,      MRRak,Wen,Rst);
DLxor2    xr2       (MEMak,        Rend,WRdn);

AsMem #(`memsize,`memdelay) mem (MRR[31:0],MRRrq,WRdn,
                                                  MAR[31:0],RDrq,WRrq,bNw,Rst);

TubeInt #`memdelay tube (Niq,Fiq,  MRR[31:0],MAR[15:13],W1,Rst);

endmodule // MemFIQ


//**********************************************************************//
//                  Asynchronous Memory (Tristate Data bus)           //
//**********************************************************************//
module AsMem(Data[31:0],Rak,Wak, Addr[31:0],Rrq,Wrq,bNw,Rst);
parameter memsize = 16384, delay = 10;

inout [31:0] Data;
output Rak,Wak;
input [31:0] Addr;
input Rrq,Wrq,bNw,Rst;
wire [31:0] Dwr,Drd;

AMem #(memsize,delay) mem (Data[31:0],Rak,Wak,Addr[31:0],Dwr[31:0],Rrq,Wrq,bNw,Rst);

MemTriDrv wr (Dwr[31:0],Data[31:0],Wrq,Wak,Rst);

endmodule


//**********************************************************************//
//                       Asynchronous Memory                          //
//**********************************************************************//
module AMem(Dout[31:0],Rak,Wak,    Addr[31:0],Din[31:0],Rrq,Wrq,bNw,Rst);
parameter memsize = 16384, delay = 10;

`define Reset_Addr  32'h00000000
`define Tube_Addr   32'h0000C000

output [31:0] Dout;
reg [31:0] Dout;
output Rak,Wak;
reg Rak,Wak;
input [31:0] Addr,Din;
input Rrq,Wrq,bNw,Rst;
reg [31:0] mask,inval,memval;

reg [31:0] mema [(memsize/4)-1:0];

`define Mem_reset begin Dout = `bx; Rak = 0; Wak = 0; end
`define Mem_undef begin Dout = `bx; Rak = `bx; Wak = `bx; end

initial
     $readmemh("Code/vbinfile", mema);

always @(Rst)
     case(Rst)
          1'b1:     `Mem_reset
          1'bx:     `Mem_undef
     endcase

always @(Rrq)
     if (!Rst)
          begin
               if (Addr < memsize)
                    #((delay*100)-150) Dout = mema[Addr>>2];
               else
                    #((delay*100)-150) Dout = 0;

               if (Dout === 32'hxxxxxxxx)
                    Dout = `h00000000;

               #150 Rak = !Rak;
          end

always @(Wrq)
     if (!Rst)
          begin
               Dout = `bz;
               #(delay*100) inval = Din;

               if (Addr < memsize)
                    begin
                         if (bNw == 1)
                              begin
                                   mask = 8'hFF << ((Addr[1:0])*8);
                                   inval = inval & mask;
                                   memval = mema[Addr>>2] & ~mask;
                                   inval = inval | memval;
                              end
                         mema[Addr>>2] = inval;
                    end

               if (((Addr & 32'h0000FFFF) != `Tube_Addr) ||
                                                  ((Din & 32'h000000FF) != `END_OF_RUN_CHAR))
               Wak = !Wak;
          end

endmodule


//**********************************************************************//
//                       Tube & Interrupt Generator                   //
//**********************************************************************//
module TubeInt(Niq, Fiq,Data[31:0], Addr[15:13], Wrq, Rst);
parameter delay = 10;

output Niq, Fiq;
reg Niq, Fiq;
input [31:0] Data;
input [15:13] Addr;
```

```
input Wrq, Rst;
reg [7:0] char;
reg [31:0] Fval, Nval;

`define Tube_reset begin Niq=1; Fiq=1; end
`define Tube_undef begin Niq='bx; Fiq='bx; end

`define END_OF_RUN_CHAR            4
`define SIM_TIME_CHAR              5
`define NS                         100

always @(Rst)
    case(Rst)
        1'b1:     `Tube_reset
        1'bx:     `Tube_undef
    endcase

always @(Wrq)
    if (Addr[15:13] == 3'b110)
        begin                                       // Tube write
            #(delay*100) char = Data & 8'hFF;
            if (char == `END_OF_RUN_CHAR)
                begin $stop; $unfreeze_waves; end
            else if (char == `SIM_TIME_CHAR)
                begin
                    $display();
                    $display("Current simulation time = %0t", $time);
                    $display();
                end
            else
                $write("%c", char);
        end

always @(Wrq)
    if (Addr[15:13] == 3'b111)
        begin
            #(delay*100) Fval = Data;
            if (Data[31] == 1)
                begin                               // FIQ interrupt
                    Fval = Data & 32'h7FFFFFFF;
                    if (Fval == 0)
                        Fiq = 1;                    // Clear FIQ interrupt
                    else
                        begin
                            $display(" *** FIQ interrupt in %0d ns ***", Fval);
                            #(Fval * `NS) Fiq = 0;
                        end
                end
        end

always @(Wrq)
    if (Addr[15:13] == 3'b111)
        begin
            #(delay*100) Nval = Data;
            if (Data[31] == 0)
                begin                               // NIQ interrupt
                    Nval = Data;
                    if (Nval == 0)
                        Niq = 1;                    // Clear NIQ interrupt
                    else
                        begin
                            $display(" *** NIQ interrupt in %0d ns ***", Nval);
                            #(Nval * `NS) Niq = 0;
                        end
                end
        end

endmodule
```