The Design and Implementation of an Asynchronous Microprocessor

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

By

Nigel Charles Paver

Department of Computer Science

1994

Table of Contents

Chapter 1 : Introduction 19
1.1 Motivation 19
1.1.1 Global synchronization 19
1.1.2 Performance 20
1.1.3 Power consumption 20
1.2 Basic concepts 21
1.2.1 Timing model 21
1.2.2 Mode 21
1.2.3 Asynchronous signalling conventions 22
1.3 Objectives and thesis structure 24
1.3.1 Structure of the thesis 24
1.3.2 Author's contribution 24
Chapter 2 : Related work 27
2.1 Automatic synthesis tools 27
2.1.1 CSP based compilation 27
2.1.2 Signal transition graphs 28
2.1.3 State machines 29
2.2 Other related work 29
2.3 Summary 30
2.3.1 Micropipelines 30
2.3.2 AMULET group Micropipelines 30
Chapter 3 : Micropipelines 31
3.1 Basic concepts 31
3.1.1 Event control modules 32
3.1.2 Metastability 33
3.1.3 Event-controlled storage element 34
3.2 Micropipelines 35
3.2.1 A Micropipeline FIFO 35
3.2.2 Micropipelines with processing 36
3.3 Event control module structures 36
3.3.1 Exclusive OR gate 37
3.3.2 Muller C-Gate 38
3.3.3 Transparent latch 39

3.3.4 SELECT block 39
3.3.5 Decision-Wait element 40
3.3.6 CALL block 41
3.3.7 TOGGLE 41
3.3.8 ARBITER 43
3.3.9 Capture-Pass latch 45
3.3.10 Cell layout 46
3.3.11 Implementation costs 46
3.4 Micropipeline implementation 48
3.4.1 The T-Latch Micropipeline 49
3.4.2 The Capture-Pass Micropipeline 51
3.4.3 Capture-Pass versus transparent latch area considerations ... 51
3.4.4 Micropipeline stage performance 53
3.4.5 Power considerations 55
3.4.6 Choosing an implementation 56
Chapter 4 : The asynchronous ARM 59
4.0.1 The ARM processor 59
4.0.2 Implementation challenges 59
4.0.3 Differences from the ARM6 60
4.0.4 Processor interface 60
4.1 Processor organization 62
4.1.1 Address interface 62
4.1.2 The register bank 62
4.1.3 The execution unit 62
4.1.4 The data interface 64
4.2 Pipeline organization and control 64
4.2.1 Dynamic pipeline structure 64
4.3 Instruction mapping 66
4.3.1 Data operation 67
4.3.2 Branch operation 68
4.3.3 Multiply operation 69
4.3.4 Load data operation 69
4.3.5 Store data operation 72
4.3.6 Block transfer operation 72
4.3.7 Exception entry operation 76
4.4 Instruction flow control 78
4.4.1 The ARM PC model and the PC pipeline 78

4.4.2 Condition code evaluation 78
4.4.3 Branch operations 79
4.5 Exception handling 81
4.5.1 Hardware interrupts 81
4.5.2 Data abort overview 83
4.5.3 Data abort signalling 84

Chapter 5 : The register bank 87

5.1 Register bank operation 87
5.1.1 Internal register structure 88
5.1.2 Register bank hazards 88
5.2 Write address storage 89
5.2.1 An asynchronous register lock FIFO 90
5.2.2 FIFO examination 91
5.2.3 Stalling reads 92
5.3 Asynchronous register bank design 92
5.3.1 Read and lock operations 92
5.3.2 Write operations 94
5.4 Additional features 95
5.4.1 Dual lock FIFO 95
5.5 Implementation 96
5.6 Future enhancements - register bypassing 98
5.6.1 Register through-passing 98
5.6.2 Last result re-use 99

Chapter 6 : Memory interface 101

6.1 Address interface 101
6.1.1 Load/Store multiple operation 103
6.1.2 Changing the PC value 103
6.1.3 PC loop deadlock 104
6.1.4 PC pipeline 105
6.1.5 Instruction overflow deadlock 105
6.1.6 PCpipe implementation 107
6.1.7 Incrementer 109
6.2 Data interface 110
6.2.1 Data in 110

Chapter 7 : Execution pipeline 113

7.1 The multiplier 113

7.2 The shifter 114
7.3 The ALU 114
7.3.1 Performance considerations 114
7.3.2 Implementation 115
7.4 The CPSR 115
Chapter 8 : Implementation 119
8.1 Design flow 119
8.1.1 Verification of the design 119
8.2 Complete organization 121
8.2.1 Datapath VLSI organization 123
8.3 Silicon layout 123
8.4 Test devices 125
Chapter 9 : Evaluation and further work 127
9.1 Design characteristics 127
9.1.1 Area overhead 128
9.1.2 Pipeline organization 129
9.2 Further work 130
9.2.1 Base technology 130
9.2.2 Processor organization 131
9.2.3 Tools 131
9.2.4 Test 132
Chapter 10 : Conclusions 133
10.1 Micropipelines 133
10.2 AMULET1 134
10.3 An asynchronous future? 134
Chapter 11 : Bibliography 135
Appendix A: Timing characteristics 145
A.1 Measurement conditions 145
SPICE timings for: CALL2 146
SPICE timings for: DMULLC2 148
SPICE timings for: DXor/DXNor 149
SPICE timings for: DWAIT2 150
SPICE timings for: MULLC2R 151
SPICE timings for: MULLC2 152

SPICE timings for: MULLC3R 153
SPICE timings for: TOGGLE 154
SPICE timings for: SELECT2 157
SPICE timings for: TLTCHR 158
Appendix B: The ARM processor 159
B.1 The ARM2 159
B.1.1 Instruction set 160
B.1.2 Organization 161
B.2 The ARM6 161

Abstract

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

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

Declaration

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

Preface

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

Acknowledgements

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

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

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

List of Figures

Figure 1-1: Two-phase communication protocol 22
Figure 1-2: Four-phase communication protocol 23
Figure 3-1: Bundled data interface 31
Figure 3-2: The two-phase bundled data convention 32
Figure 3-3: Event logic library 32
Figure 3-4: Event-controlled storage element 34
Figure 3-5: Simple Micropipeline FIFO 35
Figure 3-6: Processing Micropipeline 36
Figure 3-7: Exclusive OR gates 37
Figure 3-8: Pseudo-static Muller C-Gate 38
Figure 3-9: Transparent latch 39
Figure 3-10: SELECT block 40
Figure 3-11: Decision-Wait element 41
Figure 3-12: Call block 41
Figure 3-13: TOGGLE high-level operation 42
Figure 3-14: TOGGLE element 42
Figure 3-15: NAND gate TOGGLE 43
Figure 3-16: ARBITER implementation 44
Figure 3-17: Robust two-phase arbiter 45
Figure 3-18: Capture-Pass implementation 45
Figure 3-19: Examples of cell layout 47
Figure 3-20: Cell silicon area comparison 48
Figure 3-21: Simple transparent latch Micropipeline stage 49
Figure 3-22: Simple transmission gate latch 49
Figure 3-23: Transparent latch Micropipeline control 50
Figure 3-24: Capture-Pass Micropipeline control 51
Figure 3-25: Relative size of Capture-Pass v T-Latch Micropipeline 52
Figure 3-26: Fast-forward Micropipeline control 55
Figure 3-27: Micropipeline fork 56
Figure 3-28: Blocking Micropipeline control 57
Figure 4-1: The processor interface 61
Figure 4-2: Processor organization 63
Figure 4-3: Pipeline organization and control 65
Figure 4-4: Dynamic pipeline structure 66
Figure 4-5: Data operation datapath activity 67
Figure 4-6: Branch & branch and link datapath activity 68
Figure 4-7: Multiply datapath activity 70
Figure 4-8: Load register datapath activity 71
Figure 4-9: Store register datapath activity 73
Figure 4-10: Load/Store multiple datapath activity 75
Figure 4-11: Exception entry datapath activity 77
Figure 4-12: Instruction/PC matching 79
Figure 4-13: Interrupt and PC colour entry into the instruction stream 82
Figure 4-14: Hardware interrupt timing 83

Figure 4-15: Data abort entry into the primary decode 85
Figure 5-1: Register bank operation 87
Figure 5-2: Register bank internal structure 88
Figure 5-3: Decoder arrangement 89
Figure 5-4: Write address storage 90
Figure 5-5: Lock detection 90
Figure 5-6: Lock interrogation 91
Figure 5-7: Read lock gating 92
Figure 5-8: Register bank organization 93
Figure 5-9: A Petri Net model of the read-lock sequencing 94
Figure 5-10: Dual lock FIFO configuration 96
Figure 5-11: Register bank silicon implementation 97
Figure 5-12: Improved register write circuitry 99
Figure 6-1: The address interface 102
Figure 6-2: PC loop potential deadlock 104
Figure 6-3: Staggering of PC pipe to achieve correct R15 value 105
Figure 6-4: PC pipe behaviour across a branch 106
Figure 6-5: Potential processor deadlock 107
Figure 6-6: The PC pipelines 108
Figure 6-7: Data abort PC storage 108
Figure 6-8: Simple ripple-carry incrementer cell 109
Figure 6-9: Data-in organization 110
Figure 7-1: Execute pipeline 113
Figure 7-2: ALU carry propagation statistics 116
Figure 7-3: ARM6 v AMULET1 ALU area 117
Figure 7-4: CPSR structure 118
Figure 8-1: Design flow 120
Figure 8-2: Internal organization 122
Figure 8-3: Organization of the datapath and control dependencies 124
Figure 8-4: AMULET1 1.2 micron physical layout 125
Figure 8-5: AMULET1: 0.7 multi-project die organization 126
Figure 8-6: AMULET1: 0.7 micron multi-project die 126
Figure 9-1: Svensson style latch 131
Figure B-1: The ARM2 register organization 159
Figure B-2: Program counter and program status word 160
Figure B-3: ARM2 block diagram 162
Figure B-4: The ARM6 register organization 163

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

M.V.Wilkes.

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

Chapter 1 : Introduction

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

1.1 Motivation

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

1.1.1 Global synchronization

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

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

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

1. Very Large Scale Integration

1.1.2 Performance

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

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

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

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

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

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

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

1.1.3 Power consumption

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

1. Reduced Instruction Set Computer

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

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

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

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

1.2 Basic concepts

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

1.2.1 Timing model

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

1.2.2 Mode

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

1.2.3 Asynchronous signalling conventions

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

Two-phase

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

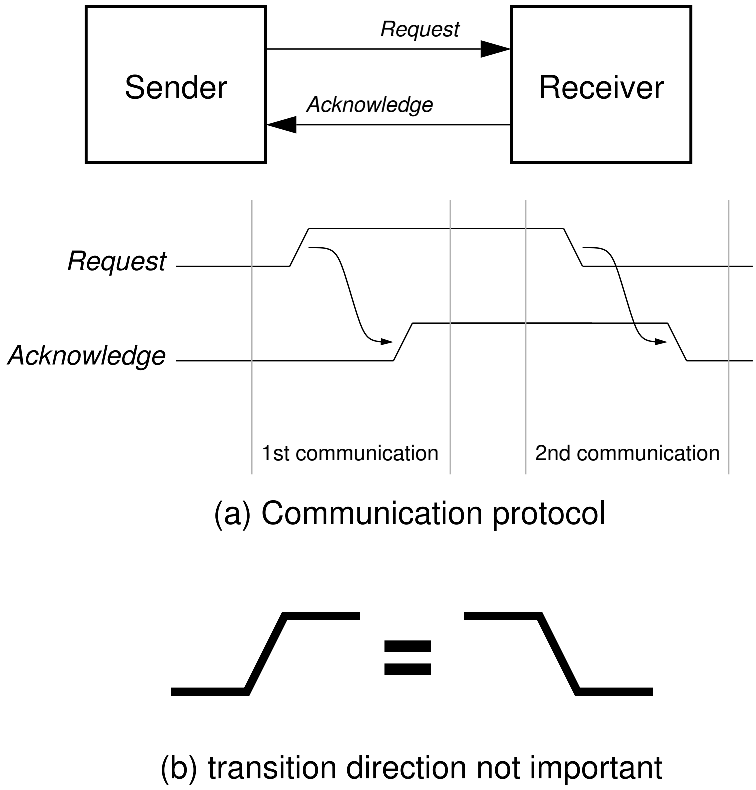


Figure 1-1: Two-phase communication protocol

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

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

Four-phase

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

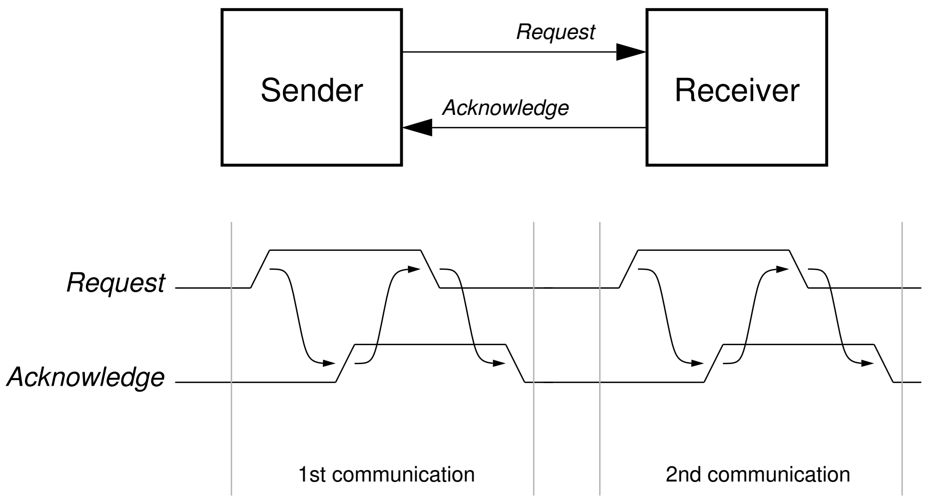


Figure 1-2: Four-phase communication protocol

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

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

Single-rail encoding

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

Dual-rail encoding

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

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

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

1.3 Objectives and thesis structure

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

1.3.1 Structure of the thesis

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

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

1.3.2 Author’s contribution

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

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

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

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

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

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

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

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

Chapter 2 : Related work

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

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

2.1 Automatic synthesis tools

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

2.1.1 CSP based compilation

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

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

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

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

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

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

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

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

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

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

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

2.1.2 Signal transition graphs

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

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

2.1.3 State machines

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

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

2.2 Other related work

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

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

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

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

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

2.3 Summary

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

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

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

2.3.1 Micropipelines

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

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

2.3.2 AMULET group Micropipelines

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

Chapter 3 : Micropipelines

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

3.1 Basic concepts

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

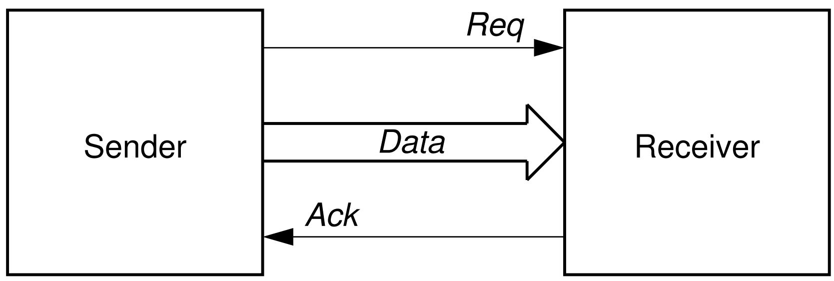


Figure 3-1: Bundled data interface

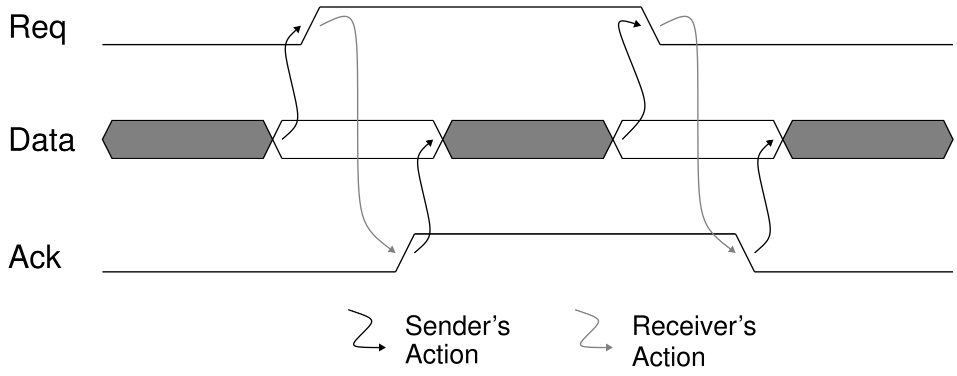


Figure 3-2: The two-phase bundled data convention

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

3.1.1 Event control modules

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

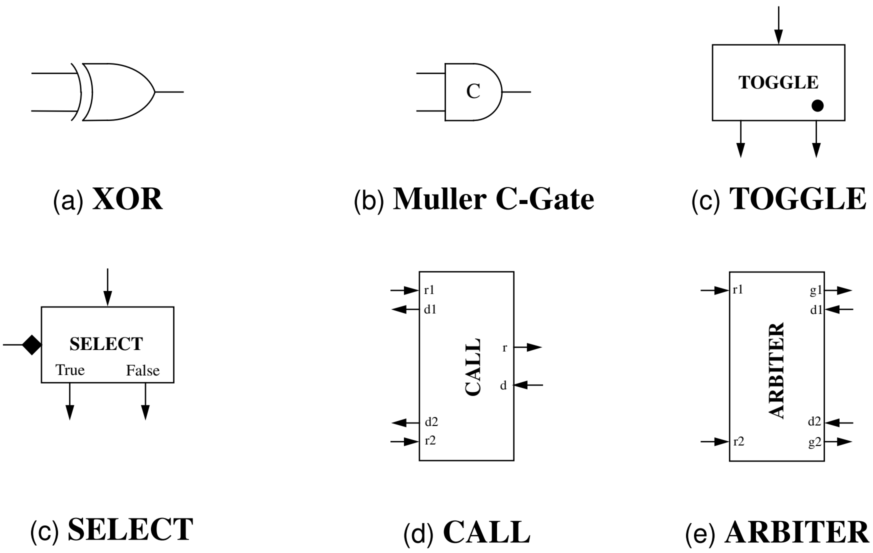


Figure 3-3: Event logic library

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

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

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

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

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

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

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

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

3.1.2 Metastability

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

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

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

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

3.1.3 Event-controlled storage element

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

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

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

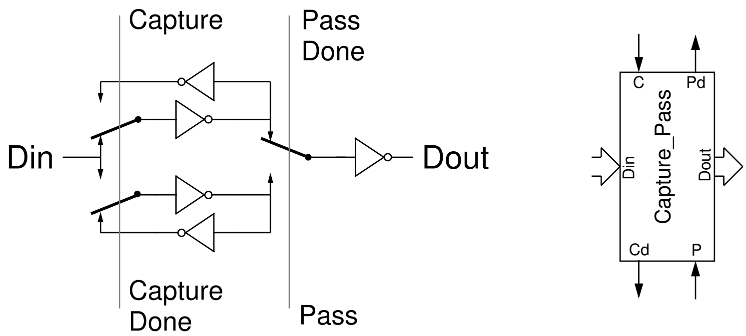


Figure 3-4: Event-controlled storage element

3.2 Micropipelines

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

3.2.1 A Micropipeline FIFO

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

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

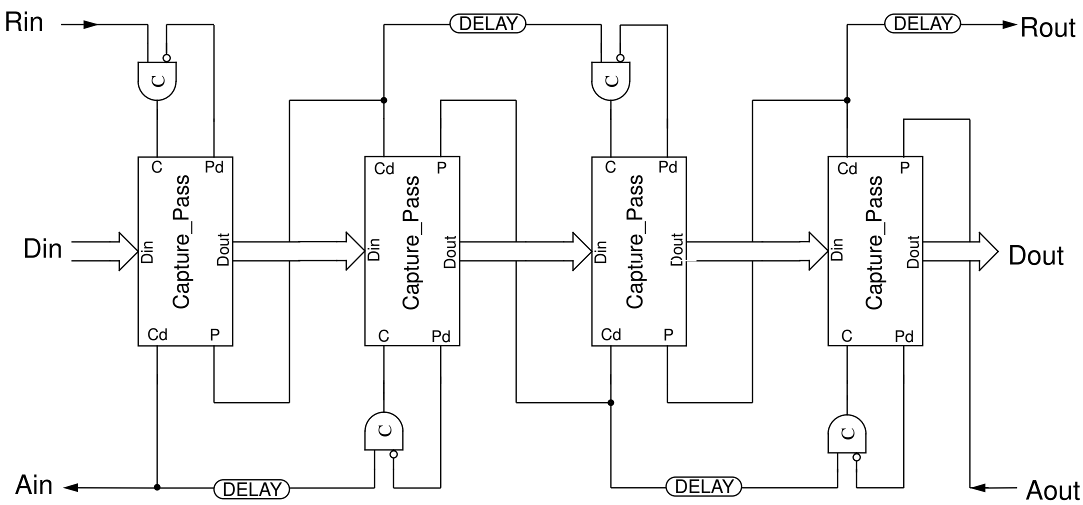


Figure 3-5: Simple Micropipeline FIFO

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

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

3.2.2 Micropipelines with processing

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

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

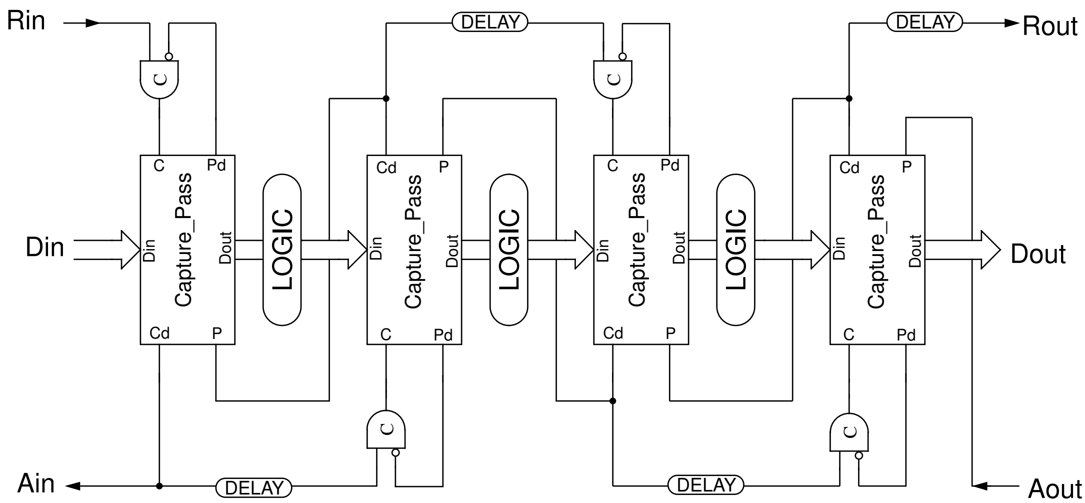


Figure 3-6: Processing Micropipeline

3.3 Event control module structures

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

3.3.1 Exclusive OR gate

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

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

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

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

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

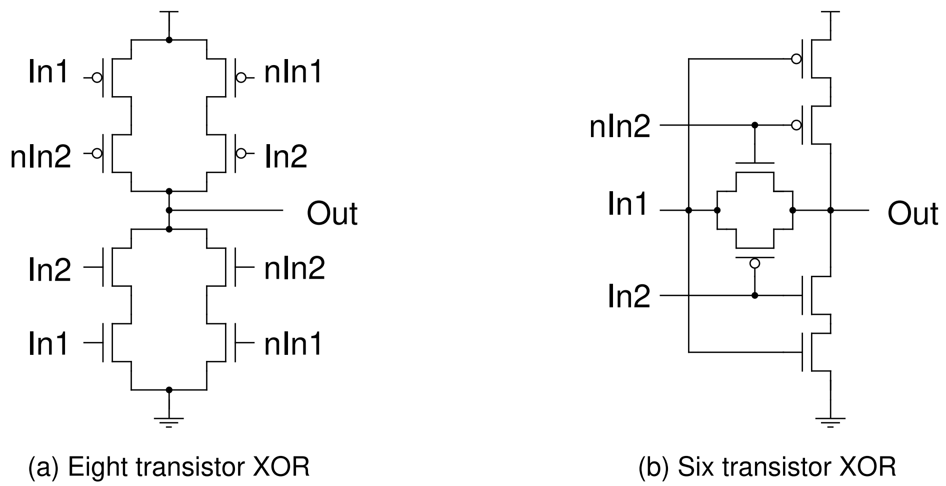


Figure 3-7: Exclusive OR gates

3.3.2 Muller C-Gate

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

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

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

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

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

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

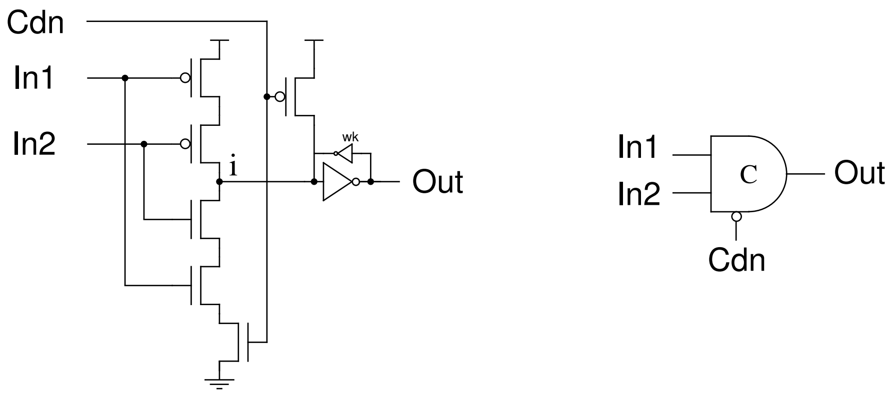


Figure 3-8: Pseudo-static Muller C-Gate

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

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

3.3.3 Transparent latch

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

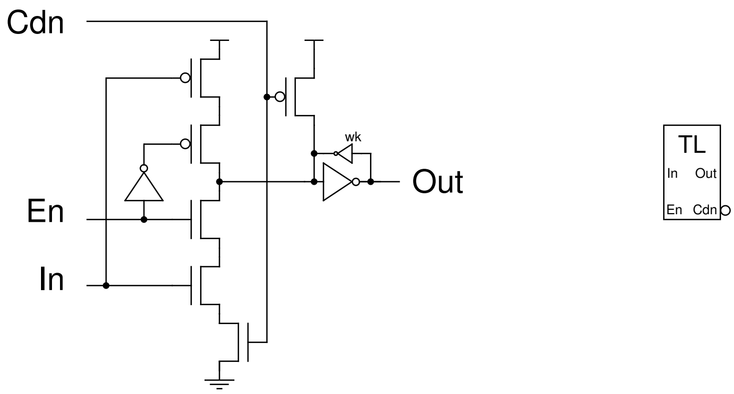


Figure 3-9: Transparent latch

3.3.4 SELECT block

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

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

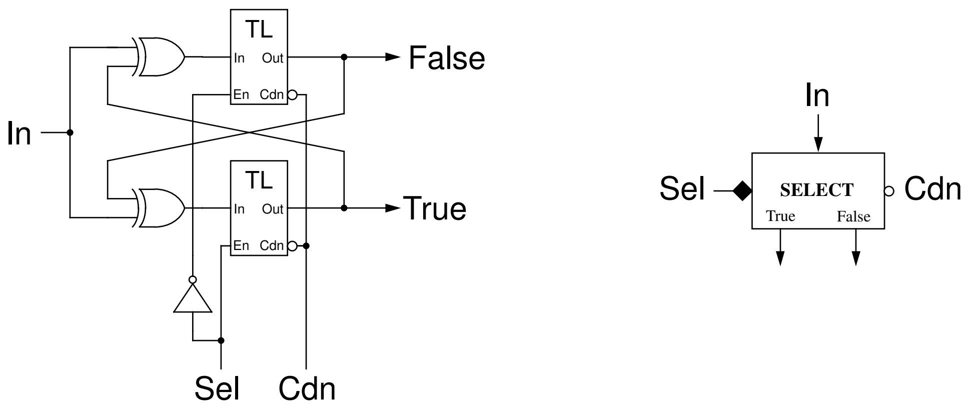


Figure 3-10: SELECT block

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

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

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

3.3.5 Decision-Wait element

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

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

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

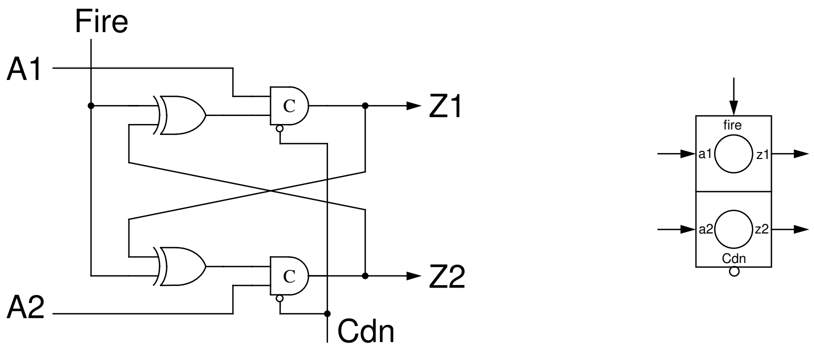


Figure 3-11: Decision-Wait element

3.3.6 CALL block

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

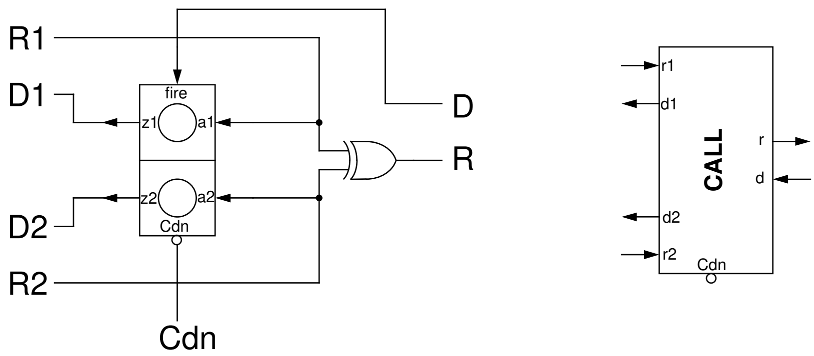


Figure 3-12: Call block

3.3.7 TOGGLE

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

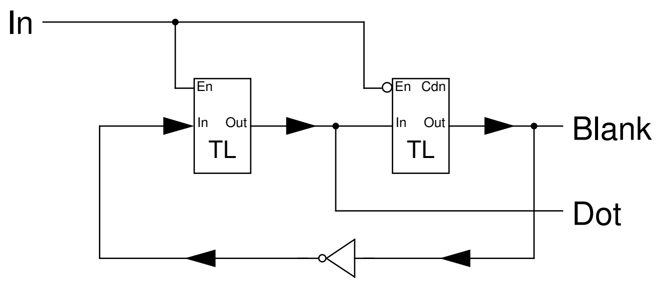


Figure 3-13: TOGGLE high-level operation

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

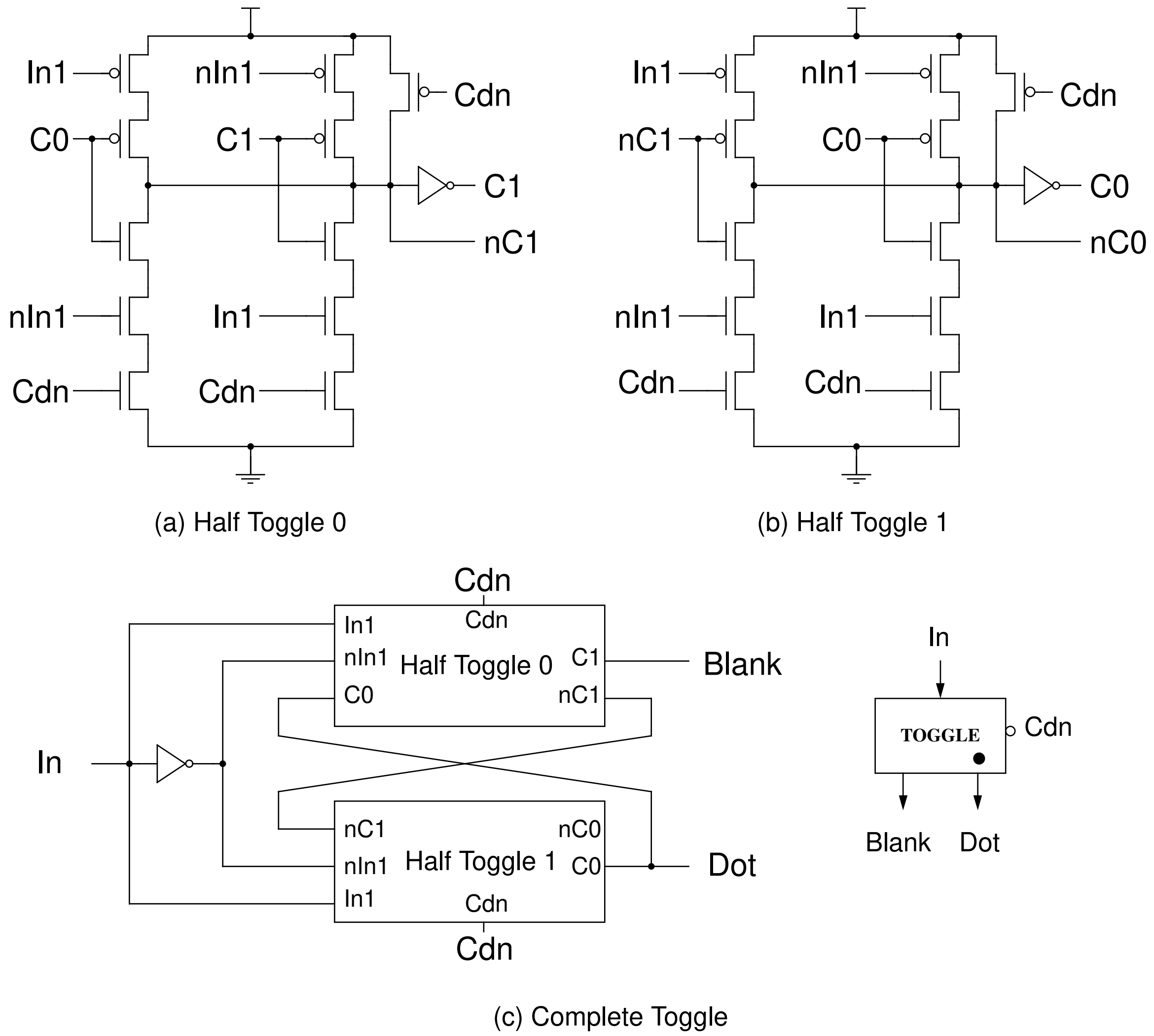


Figure 3-14: TOGGLE element

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

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

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

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

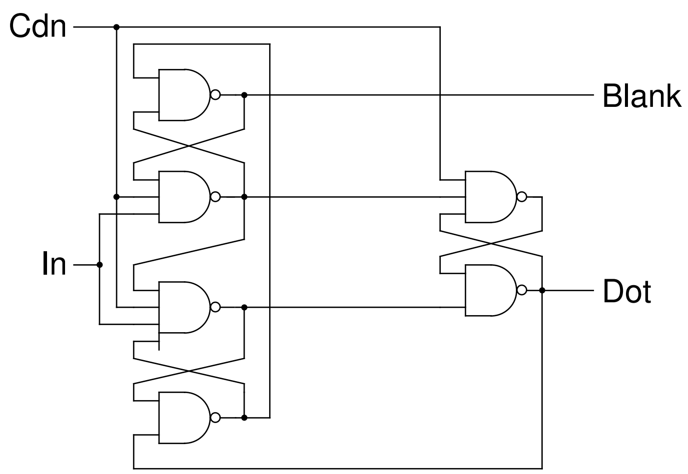


Figure 3-15: NAND gate TOGGLE

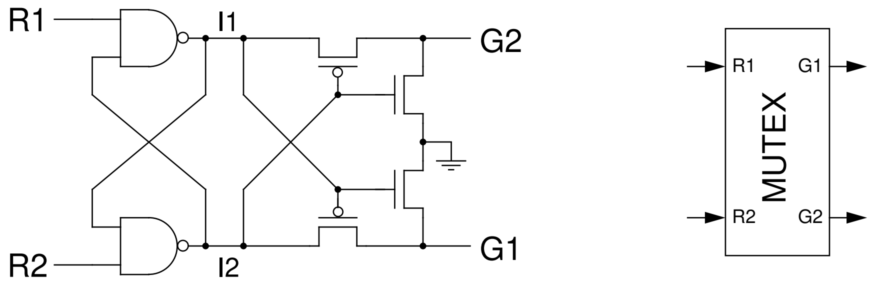
3.3.8 ARBITER

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

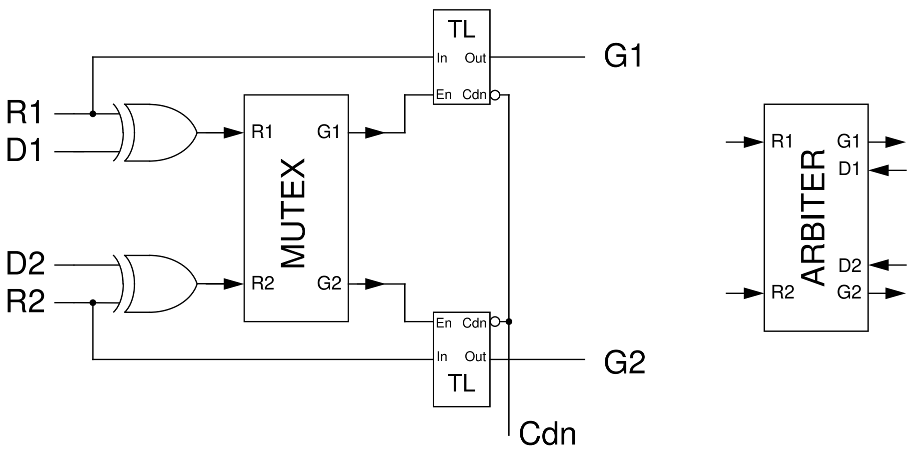
If both inputs go high at the same time the R-S flip-flop may go into a metastable state where both internal nodes remain at an intermediate, (non-digital level) for an arbitrary amount of time. With intermediate levels on the internal nodes the output inverters are unable to turn on and pull either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

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

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



(a) Four-phase mutual exclusion circuit



(b) Two-phase Micropipeline arbiter

Figure 3-16: ARBITER

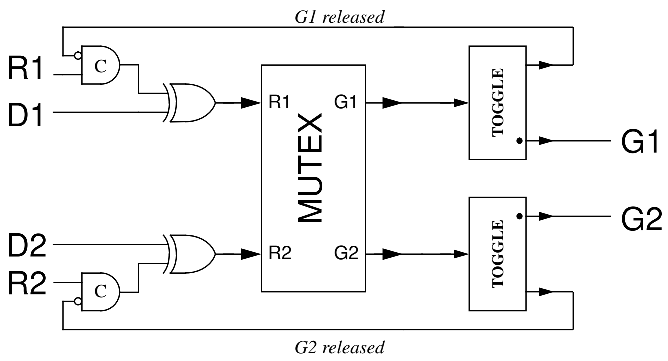


Figure 3-17: Robust two-phase arbiter

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

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

3.3.9 Capture-Pass latch

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

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

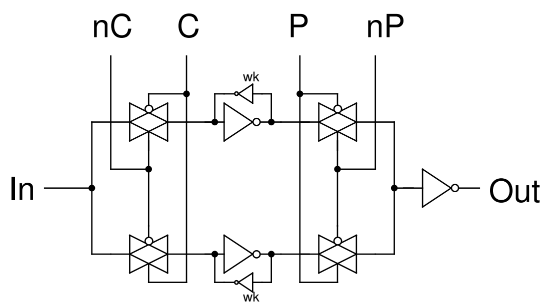


Figure 3-18: Capture-Pass implementation

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

3.3.10 Cell layout

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

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

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

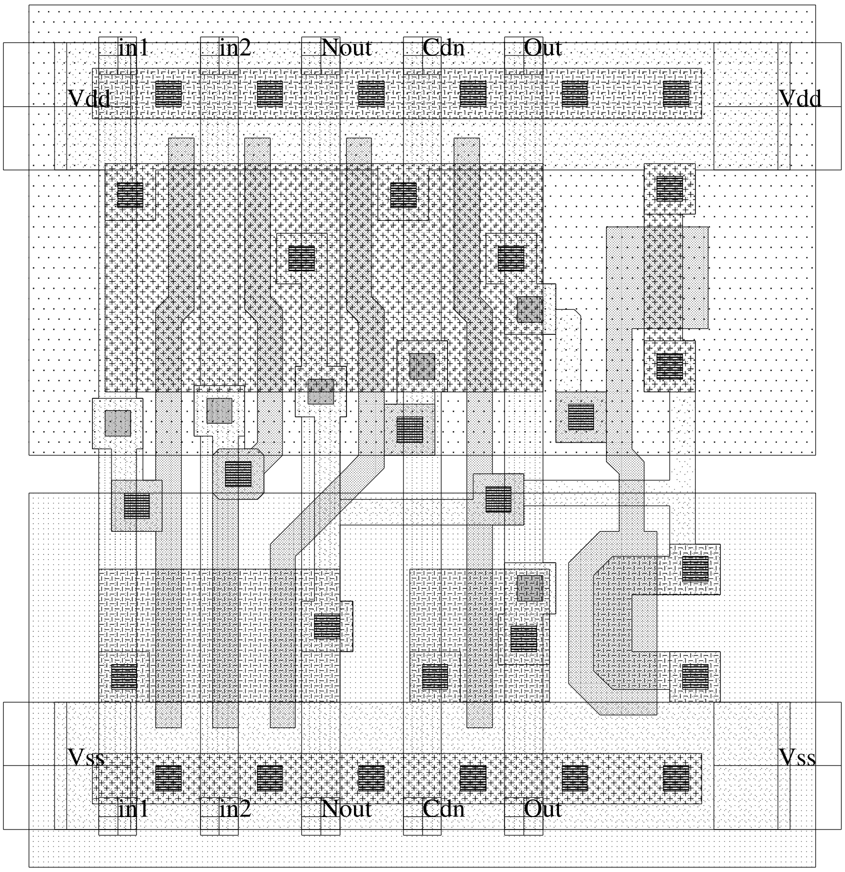
The layout cells were extracted and their operation analysed with SPICE under varying load conditions. The resulting timing characteristics can be found in appendix A.

3.3.11 Implementation costs

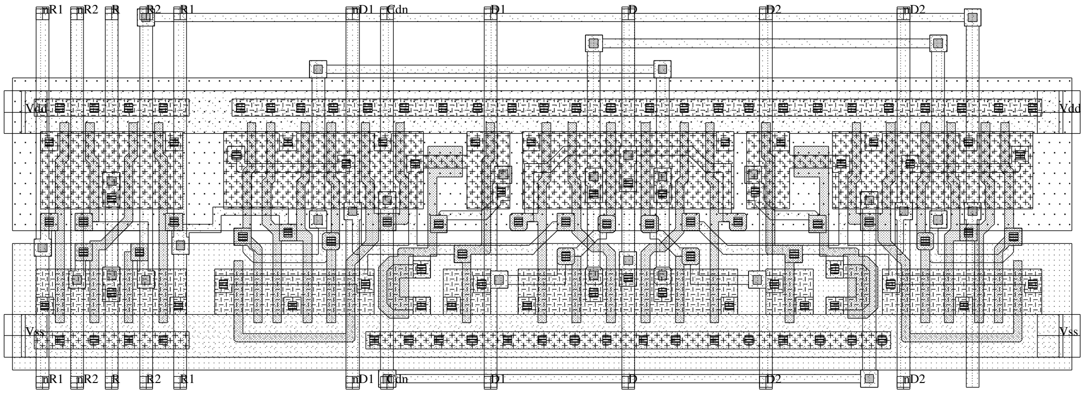
Figure 3-20 shows a plot of the relative sizes of the asynchronous cells taken from the layout editor of the Compass Design Automation VLSI design tools [Comp91a]. There is a single strength inverter (**Inv1**) on the right to act as a scale reference point.

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

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



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



(b) Layout of a CALL block

Figure 3-19: Examples of cell layout

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

3.4 Micropipeline implementation

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

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

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

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

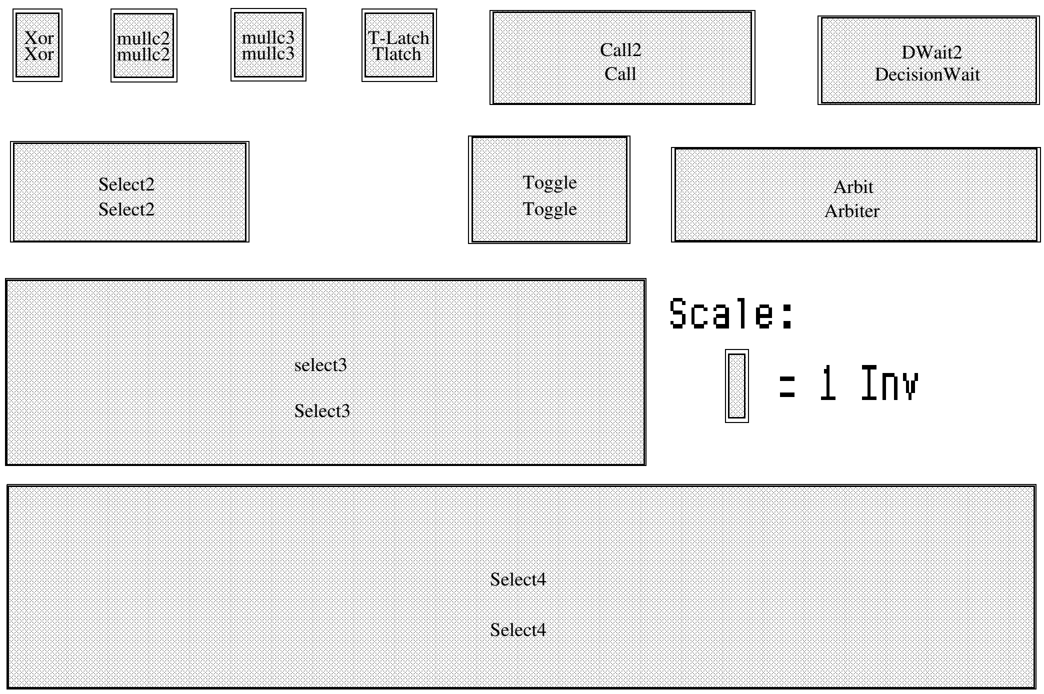


Figure 3-20: Cell silicon area comparison

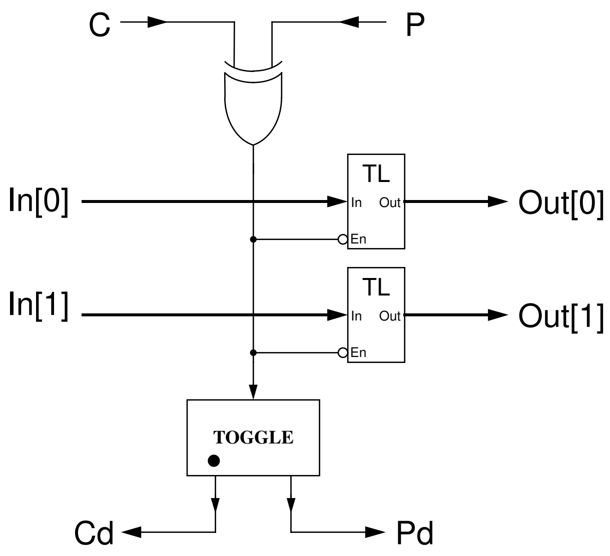


Figure 3-21: Simple transparent latch Micropipeline stage

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

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

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

3.4.1 The T-Latch Micropipeline

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

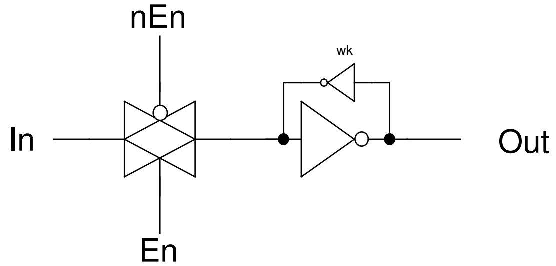


Figure 3-22: Simple transmission gate latch

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

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

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

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

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

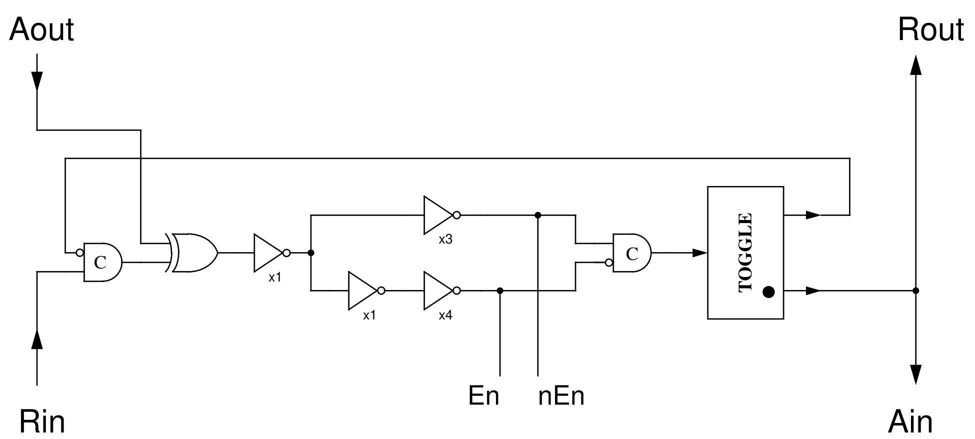


Figure 3-23: Transparent latch Micropipeline control

3.4.2 The Capture-Pass Micropipeline

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

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

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

3.4.3 Capture-Pass versus transparent latch area considerations

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

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

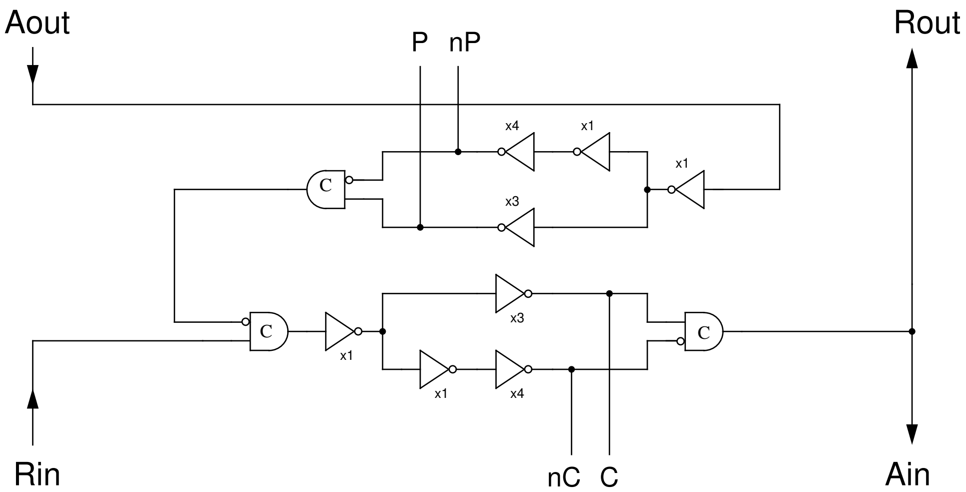


Figure 3-24: Capture-Pass Micropipeline control

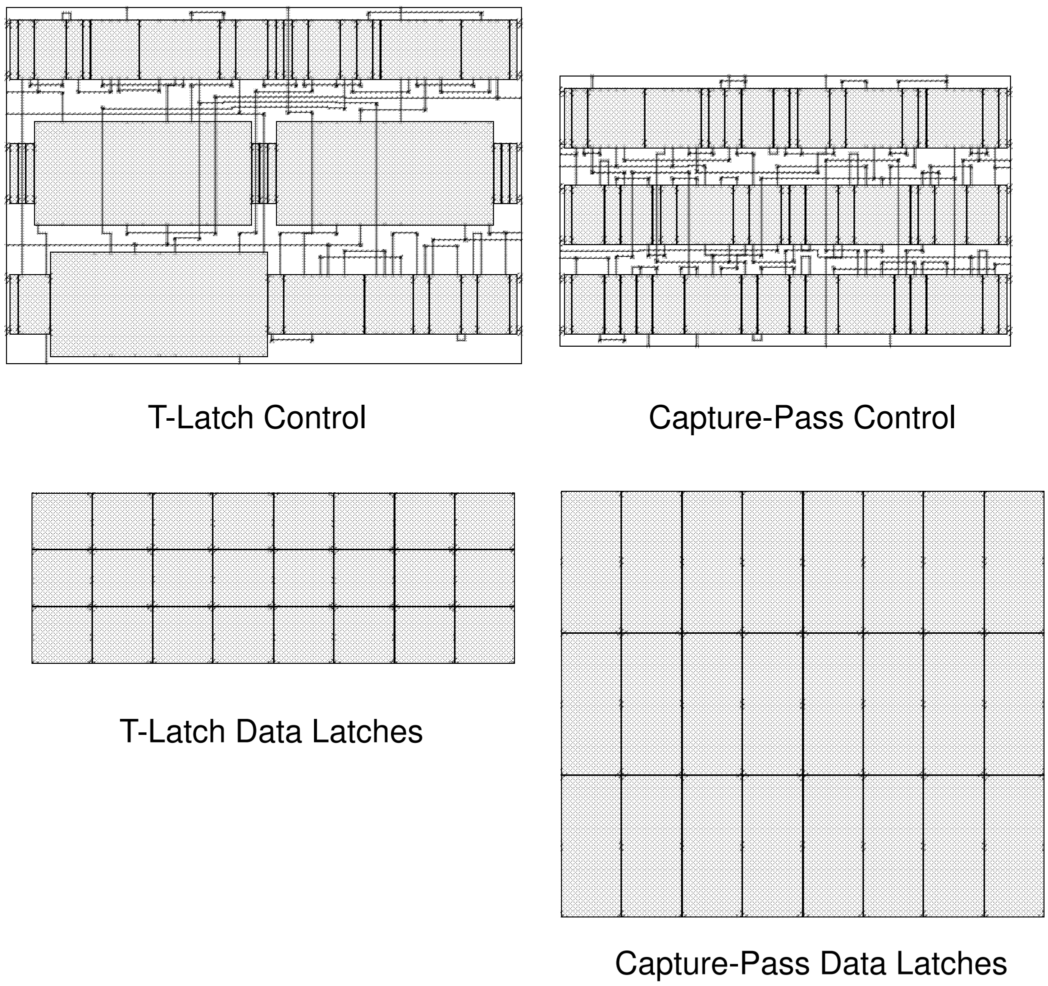


Figure 3-25: Relative size of Capture-Pass v T-Latch Micropipeline

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

Table 1:

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

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

Note: for the VLSI CMN12 process λ=0.6 Microns

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

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

Table 2:

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

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

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

3.4.4 Micropipeline stage performance

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

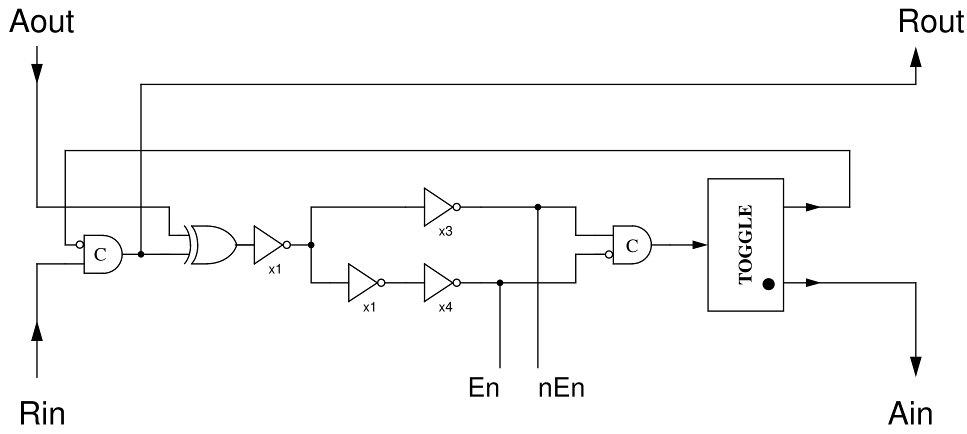


Figure 3-26: Fast-forward Micropipeline control

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

3.4.5 Power considerations

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

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

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

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

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

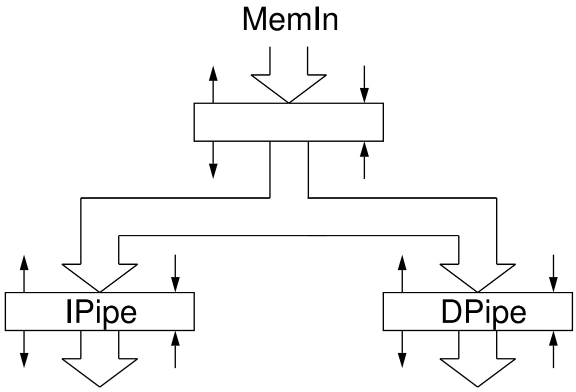


Figure 3-27: Micropipeline fork

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

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

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

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

3.4.6 Choosing an implementation

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

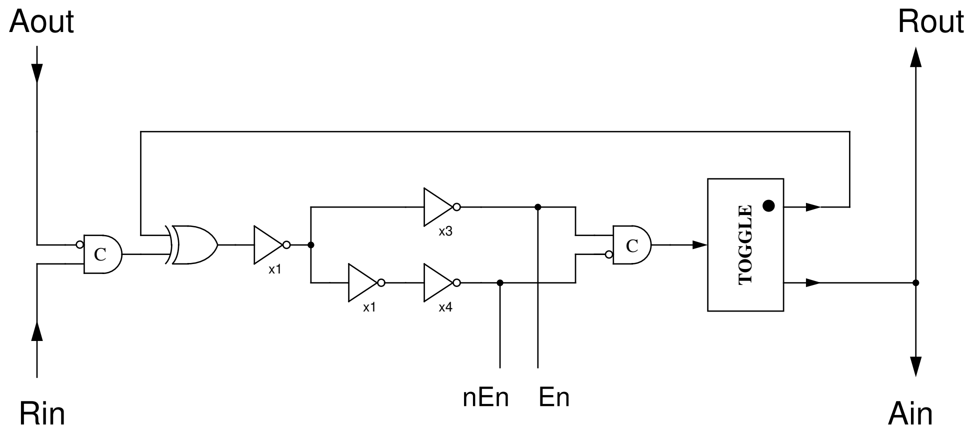


Figure 3-28: Blocking Micropipeline control

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

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