



Fast Asynchronous VLSI Circuit Design Techniques and their Application to Microprocessor Design

Shannon V. Morton, B. E. (Hons.)

Department of Electrical and Electronic Engineering
The University of Adelaide
Adelaide, South Australia.

January 22, 1997

Addenda

Section 2.2 - Asynchronous hardware

This thesis is focussed on practical asynchronous circuit design with an emphasis on micro-processors, and therefore only those processors which have been designed to fabrication have been included in the literature discussion. Fabricated test structures and coded microprocessors do not provide enough reliable data to justify their discussion in this context, although in Section 8.1 a description of coded superscalar processors, as compared to the author's, is given.

Section 2.2.2 - AMULET I and II

The AMULET I design was the first generation attempt at implementing a sixth generation commercial ARM6 processor, and was built with significantly less man-power and resources. This makes comparisons between them difficult, and the performance gap of 50% should be treated cautiously. The subsequent AMULET II processor has since demonstrated improved performance over the ARM6.

Section 3.5 - The ECS representation

The ECS representation is intended to enable asynchronous circuits to be specified in a clear and concise format which models the interaction of data and control wires. It is not intended as a formal tool for synthesis, and as such has not been developed using formal methods. Instead, an intuitive description of the representation has been presented based on the practical implementation of asynchronous circuits, as this is the major focus of the thesis.

Section 4.1 - Algebraic improvements of a TS

The simplifications described in this section are synonymous with those of boolean logic.

Section 5.3.1 - Dynamic logic

The nature of the data stream will also impact the power dissipation of a dynamic versus a static gate.

Section 5.3.3.2 - Self-timed pseudo-nmos logic

This circuit has a fast completion detection time compared to the static logic tree, as evidenced in Table 6.8. Compared to a typical dynamic gate however, it will be slow.

Chapter 6 - Self-timed Architectures

The pseudo self-timed architectures presented in this chapter do not require additional safety margins. The computation and completion paths are closely matched in layout, and an implicit margin is already included in the handshaking overhead to compensate for any variations.

Errata

Page 47, line 1: "sinks input" should read "sink's input"

Page 77, line 2: "though" should read "through"

Page 86, line 3: "is" should read "are"

Page 99, line 29: "need" should read "needed"

Page 123, line 11: "best the there" should read "best there"

Page 155, line 11: "blocks" should read "block"

Contents

Abstract	ix
Declaration	x
Preface	xi
Acknowledgements	xii
List of Figures	xiii
List of Tables	xvi
1 Introduction	1
1.1 Advantages of asynchronous systems	2
1.1.1 Global communication	2
1.1.2 Data dependent computation times	3
1.1.3 Resilience to operating conditions	3
1.1.4 Reduced power dissipation	4
1.1.5 Incremental improvements	4
1.1.6 Synthesis and verification	4
1.1.7 Power spectrum	5
1.2 Disadvantages of asynchronous systems	5
1.2.1 Control complexity	5
1.2.2 Testability	5
1.2.3 Area overhead	6
1.2.4 Operating speed	6
1.2.5 Integration and software support	7

1.3	Asynchronous paradigms	7
1.3.1	Timing assumptions	7
1.3.2	Control signalling	8
1.3.3	Signal encoding	9
1.3.4	Summary	10
1.4	Thesis outline	10
2	Related Work	12
2.1	Synthesis and verification	12
2.1.1	Tangram	13
2.1.2	Communicating processes	13
2.1.3	Signal transition graphs (STGs)	14
2.1.4	STG related synthesis	15
2.1.5	Other contributions	15
2.2	Asynchronous hardware	15
2.2.1	Micropipelines and the CFPP	16
2.2.2	AMULET I and II	17
2.2.3	DCC error detector	17
2.2.4	Caltech microprocessor	18
2.2.5	Other contributions	18
2.3	Summary of related works	18
2.4	A need for speed	19
3	Event Controlled Systems (ECS) Design Representation	21
3.1	Conventions	21
3.2	Event controlled elements	22
3.2.1	Muller-C element	22
3.2.2	Merge gate	23
3.2.3	Send gate	24
3.2.4	Feed gate	25
3.2.5	Restore gate	26
3.2.6	Until gate	27
3.2.7	Latching element	28

3.2.8	Delays and logic functions	29
3.2.9	Relative gate speeds	29
3.2.10	Micropipeline module exceptions	29
3.3	Some formalisms for gate representations	30
3.4	Analysis of methodologies	31
3.5	The ECS representation	33
3.5.1	Transforming signals into the temporal domain	34
3.5.2	ECS operators and temporal equations	36
3.5.3	Some ECS gate examples	40
3.5.4	Comparative gate representations	43
3.5.5	Some example TS's and their corresponding circuits	43
3.5.6	Precedence and properties of temporal operators	45
3.5.7	Interconnectivity of gates in ECS	46
3.5.8	Principles of error detection	47
3.6	Summary	50
4	Fast Asynchronous Circuit Techniques	51
4.1	Algebraic improvements of a TS	51
4.1.1	Useless TE substitutions	54
4.1.2	Useful TE substitutions	55
4.1.3	Taking advantage of the <i>typical</i> scenario	55
4.2	Improving acknowledge times	56
4.2.1	Example: sharing of a common unit	59
4.2.2	Example: data latching circuits	60
4.2.3	Comments on improving acknowledgements	63
4.3	Activating functional units	64
4.3.1	Conditionally activated parallel units	64
4.3.2	Generating a ∂out event in the general sense	66
4.3.3	Generating a ∂out event for exclusively triggered units	68
4.3.4	Splitting a tree of <i>select</i> gates into individual <i>feed</i> gates	69
4.4	Reducing event path delays	69
4.4.1	Moving metastability detection out of the event path	69

4.5	Summary	73
5	Asynchronous Pipelines	74
5.1	FIFO pipelines	75
5.1.1	Micropipeline 2P FIFOs	75
5.1.2	4P FIFO circuits	76
5.1.3	A fast ECS FIFO	77
5.1.4	Comparison of FIFO designs	79
5.2	Pipelines with processing delays	80
5.3	Precharge pipelines: general concepts	82
5.3.1	Dynamic Logic	83
5.3.2	Requirements of a PP for dynamic logic	84
5.3.3	Methods of completion and precharge detection	86
5.4	Decoupled 4P precharge pipelines	90
5.4.1	Implementations for $PP\alpha$, $PP\beta$, and $PP\gamma$	90
5.4.2	Performance comparisons	91
5.5	ECS precharge pipelines	92
5.5.1	$PP\alpha$ implementation	92
5.5.2	$PP\beta$ implementation	93
5.5.3	$PP\gamma$ implementation	94
5.5.4	Performance comparisons	95
5.6	Comparison of ECS and D4P PP structures	96
5.7	Summary	97
6	Self-Timed Architectures	98
6.1	Strict self-timing requirements	99
6.2	Designing and utilizing self-timed units	101
6.3	Adder Structures	102
6.3.1	Self-timed ripple carry implementation	103
6.3.2	Self-timed ripple select implementation	105
6.3.3	Comparison of ST adders	106
6.3.4	Pseudo self-timing (PST)	107
6.3.5	PST ripple carry implementation	108

6.3.6	PST ripple select implementation	109
6.3.7	Comparison of PST and ST adders	110
6.4	Incrementer structures	112
6.4.1	Self-timed incrementer	113
6.4.2	Incrementer performance	114
6.5	Comparator structures	115
6.5.1	Possible implementations	116
6.5.2	Comparator tree	117
6.5.3	Comparator performance	119
6.6	Multiplier structures	120
6.6.1	Exploiting self-timed operation	121
6.6.2	Simple partial product generation	121
6.6.3	Radix 4 Booth encoding for generating partial products	122
6.6.4	Recoding Booth's algorithm to improve performance	124
6.6.5	Implementation, floorplanning, and area usage	125
6.6.6	Performance and comparisons	126
6.6.7	Potential improvements	128
6.7	Summary	130
7	<i>ECSTAC</i>: A Pipelined Microprocessor	131
7.1	Design considerations	132
7.2	Instruction set architecture	133
7.3	Architectural overview	135
7.4	Processor sub-systems	137
7.4.1	Instruction decode	137
7.4.2	Operand fetch	138
7.4.3	Adder, comparator, and stack processor	142
7.4.4	Arithmetic and logical unit	147
7.4.5	Order unit	148
7.4.6	Registers and scoreboarding	149
7.4.7	Program counter	151
7.5	Testability issues	154

7.5.1	Delay modelled V_{tt} bus	154
7.5.2	Interface delays	155
7.5.3	Scan testing	155
7.6	Simulation results	156
7.6.1	Sub-system simulations	157
7.6.2	Core simulation environments	158
7.6.3	General purpose instruction streams	159
7.6.4	Instruction streams for determining bottlenecks	161
7.6.5	Comparisons	162
7.7	Summary	164
8	<i>ECSCCESS</i>: A Superscalar Microprocessor	166
8.1	Other asynchronous superscalar microprocessors	167
8.1.1	SCALP	167
8.1.2	Fred	168
8.1.3	Rotary pipeline processor	169
8.2	Characteristics of <i>ECSCCESS</i>	170
8.3	Instruction set architecture	171
8.4	General architecture	173
8.5	Implementation of the <i>shore</i>	174
8.5.1	Controlling RAW hazards	175
8.5.2	Controlling WAR hazards	176
8.5.3	Structure of the pre-FU unit	177
8.5.4	Generating the return event to the sun	178
8.5.5	Switching network	179
8.6	Implementation of the <i>sun</i> and <i>moons</i>	179
8.6.1	Globe controller	180
8.6.2	PC controller	181
8.6.3	Branch moon controller	182
8.6.4	Stack moon controller	182
8.7	Implementation of functional units	183
8.7.1	AID unit	183

8.7.2	MEM unit	184
8.7.3	CMP unit	184
8.8	Floorplanning issues	185
8.8.1	Size of the ocean	185
8.8.2	Size of the switching network	185
8.8.3	A floorplan based on the minimum FU width	186
8.8.4	Floorplanning for a larger FU width	187
8.9	Simulation results	188
8.10	Comparisons	190
8.11	Extensions and improvements	191
8.11.1	Incorporating interrupts	191
8.11.2	Exception handling	193
8.11.3	Reducing the ocean width for WAR and RAW hazards	194
8.12	Summary	195
9	Conclusions	196
9.1	Further work	199
A	Fundamental Temporal Equations and Corresponding ECS Gates	201
B	ISA of the <i>ECSTAC</i> Microprocessor	203
B.1	Memory instructions	203
B.1.1	Two byte instructions	203
B.1.2	Four byte instruction	204
B.1.3	The unused mode	204
B.2	ALU instructions	204
B.2.1	Two byte instruction (short mode)	204
B.2.2	Three byte instructions (long mode)	205
B.3	Branch instructions	205
B.3.1	One byte instruction - CALL	205
B.3.2	Two byte instructions - BRANCH	206
B.4	Stack instructions	206
B.5	Special instructions	207

C	ISA of the <i>ECSCCESS</i> microprocessor	208
C.1	Branch instructions	208
C.2	Interrupt instructions	209
C.3	MOVE instruction	210
C.4	LDC instruction	210
C.5	FU instructions	211
C.5.1	Register unit	211
C.5.2	Arithmetic unit	212
C.5.3	Multiply, divide, and sqrt unit	212
C.5.4	Shifter and logical unit	213
C.5.5	Comparator unit	213
C.5.6	Memory unit	214
C.5.7	Floating point units and co-processors	215
	Bibliography	216

Abstract

Over the past decade a variety of asynchronous synthesis techniques have been proposed. The majority of these have been concerned with generating provably correct circuits with high reliability, whereas others have focussed on producing circuits with low power dissipation. However in taking such approaches the resulting circuits are usually swamped with a large number of gates in the critical paths and are consequently inefficient in terms of speed.

This thesis describes a collection of novel design techniques engineered for high speed operation (such as fast pipeline control circuits and pseudo self-timed computations). In addition, a new gate representation is proposed to better reflect their functionality in an asynchronous domain. As an illustration of these design techniques two microprocessors have been implemented:

- *ECSTAC* is styled as a linear pipeline with a load/store architecture and an 8 bit data path and a 24 bit address path. It employs fast pipeline control circuits and utilizes some interesting asynchronous techniques for bypassing stages, controlling data hazards, and register fetching. *ECSTAC* has been fabricated using ES2's 0.7 μ m DLM CMOS process and demonstrated a peak operating speed of 28 Mips.
- *ECSCCESS* is structured to take advantage of self-timed data dependent computations and to employ functional parallelism. It has a 32 bit data path and can provide for up to 32 single precision (16 double precision) functional units which interact directly with each other, thus enabling out-of-order execution and global results forwarding. Their operation is fully decoupled from branches and interrupts to minimize stalling. Emphasis has been placed on maintaining a high throughput to the functional units. It employs novel design techniques for rapid data hazard detection between units, PC updating, and decoupled branch evaluation and branch target determination. *ECSCCESS* has been simulated in VHDL with delays comparable to those of the 0.7 μ m standard cell library used in *ECSTAC*, and demonstrated a peak operating speed of 181 Mips.

Declaration

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

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

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

S.V. Morton

January 22, 1997

Preface

The author obtained his degree in Electrical & Electronic Engineering at the University of Adelaide in 1991, graduating with first class honours as *dux* of the class. He then enrolled in the degree of Doctor of Philosophy at that same university in 1992 of which this thesis is the culmination. The author also took one year's intermission to work in England on the EXACT project which involved a collaboration between a number of European companies and universities investigating asynchronous systems.

S.V. Morton

January 22, 1997

Acknowledgements

I would firstly like to thank my partner in crime Sam S. Appleton with whom I've worked on this project for the last few years, and who kept himself out of trouble by designing a beastly fast cache system for the *ECSTAC* processor (so if it doesn't work, it's his fault) as well as numerous other groovy control structures. He's been a great man to bounce ideas off (as well as rubber balls) and a pretty good raver to boot. Thanks Sammy.

A special thanks too to my supervisor, Mike Liebelt, who unlike his stunt double "Andrew Denton" has not treated the project as comic relief. His guidance and expertise has kept me on the bright, technicolour highway and out of the dark alleyed dead-ends I'd have probably trod without him. He's also been great at getting us well-needed funds for this research, as well as being very understanding of my nocturnal instincts.

I'd also like to thank those others who have come and gone along the way: Dr. Pucknell for his pioneering work in this field and some rather humourous final year lectures; Andrew Johnson for helping the project get off the ground to start with; Jungwook Yang for his initial work in devising a suitable cache architecture; and all the final year students who got the best projects ever in the known *and* unknown universes to work on.

Let me also thank the English connection. In particular Dr. Mark Josephs of Southbank University in London who gave me the opportunity to work with their group for a year (as well as giving me lots of pounds), and the whole AMULET crew of Manchester University whose brilliance in this field is truly admirable. Even the fat one.

Lastly, let me thank my parents for !everything!, the guys in the HiPCAT and VLSI Labs for letting me play groovy music and exchanging filthy comments, and the two most wonderful women in the world (Debz and Joolz) for lots of other stuff...

S.V. Morton (always jammin')

List of Figures

1.1	The clock skew problem and an alternative asynchronous protocol.	2
1.2	The isochronic fork.	8
1.3	Four and two phase handshaking protocols.	9
2.1	Signal transition and state graphs for the <i>cgate</i>	14
2.2	A micropipeline architecture with a logic-level latch controller.	16
3.1	Symbol, timing, and VLSI layout of a <i>cgate</i>	22
3.2	Symbol and timing of a <i>merge</i> gate.	23
3.3	VLSI layouts for an <i>xor</i> gate.	24
3.4	Symbol and timing of a <i>send</i> gate.	24
3.5	VLSI layouts for a <i>latch</i>	25
3.6	Symbol, timing, and circuit diagram of a <i>feed</i> gate.	26
3.7	Symbol of a <i>select</i> gate.	26
3.8	Symbol and timing of a <i>restore</i> gate.	26
3.9	Circuit diagrams for the <i>restore</i> gate.	27
3.10	Symbol and timing of the <i>until</i> gate.	28
3.11	Two implementations of an event driven <i>latch</i>	29
3.12	Representations of the <i>cgate</i> , <i>send</i> , and <i>feed</i> gates using STGs.	32
3.13	A typical transitioning sequence of a gate in ECS.	36
3.14	Waveforms for the <i>send</i> and <i>feed</i> gates in the voltage and temporal domains.	41
3.15	Waveforms for the <i>cgate</i> and a <i>latch</i> in the voltage and temporal domains.	42
3.16	An ECS reconfigurable FIFO.	44
3.17	A slice of the input circuitry for a discrete Fourier transform unit.	44
3.18	An illustration of a ϕ event line.	46
3.19	Examples of EV errors for the <i>send</i> and <i>merge</i> gates.	48

3.20	Determining a CV error using rise and fall times.	49
3.21	Example of a potential glitch for the <i>and</i> gate.	49
4.1	Simple gated pulse circuit and its temporal specification.	52
4.2	Degraded gated pulse circuit and its temporal specification.	53
4.3	Improved gated pulse circuit and its temporal specification.	54
4.4	An example system constructed in a SI environment.	57
4.5	An improved ECS version of the example system.	58
4.6	SI and ECS circuits for sharing a common unit.	59
4.7	Two SI micropipeline control structures for the data latching circuit.	60
4.8	An ECS implemented SI version of the data latching circuit.	61
4.9	Two optimized ECS implementations of the data latching circuit.	62
4.10	Conditional activation of unit <i>Y</i> in parallel with unit <i>X</i>	64
4.11	Two SI implementations for generating a <i>∂out</i> event.	65
4.12	An ECS implementation for generating a <i>∂out</i> event.	66
4.13	A generalized conditional trigger structure using SI and ECS approaches.	67
4.14	Generating <i>∂out</i> for exclusively triggered units using SI and ECS approaches.	68
4.15	A halting circuit with metastability and its transfer characteristic.	70
4.16	An improved halting circuit with metastability and its transfer characteristic.	71
4.17	An Hspice simulation of the improved halting circuit with metastability.	72
5.1	A micropipeline stage also indicating a <i>fast-forward</i> implementation.	76
5.2	A simple four phase FIFO controller.	77
5.3	A decoupled four phase FIFO controller.	78
5.4	An ECS micropipeline and the state pipeline FIFO controller.	78
5.5	A typical delay-modelled pipeline.	80
5.6	A delay element for positive transitions only.	81
5.7	A general dynamic logic computational block.	83
5.8	A self-timed static logic method for generating <i>cdone</i> and <i>pdone</i>	86
5.9	A self-timed pseudo-nmos logic method for generating <i>cdone</i> and <i>pdone</i>	87
5.10	A method for generating <i>cdone</i> and <i>pdone</i> which closely models the worst case pull-down time.	88
5.11	A delay modelled method for generating <i>cdone</i> and <i>pdone</i>	89

5.12	Precharge pipelines implemented with a decoupled four phase controller.	91
5.13	An α precharge pipeline implemented in ECS.	93
5.14	A β precharge pipeline implemented in ECS.	93
5.15	A γ precharge pipeline implemented in ECS.	95
6.1	A self-timed ripple carry implementation of an adder cell.	104
6.2	A self-timed ripple select implementation of an adder cell.	106
6.3	Self-timed and pseudo self-timed generalized views of an adder cell.	107
6.4	A pseudo self-timed ripple carry implementation of an adder cell.	108
6.5	A pseudo self-timed ripple select implementation of an adder cell.	109
6.6	The adder cell and validity detection used in the AMULET processor.	111
6.7	A self-timed incrementer without unnecessary carry propagations.	114
6.8	A 2 bit comparator node and the generation of its initial inputs.	118
6.9	Symmetric and asymmetric tree structures for a full comparator.	118
6.10	Configuration of a self-timed multiplier.	126
6.11	Two possible floorplans of a self-timed multiplier.	127
6.12	A low area implementation of a self-timed multiplier.	129
7.1	The general structure of the <i>ECSTAC</i> microprocessor.	135
7.2	A block diagram of the OF stage.	138
7.3	Control circuit for routing the data from the ID into the OF stage.	139
7.4	Event bypass method for controlling register accesses.	141
7.5	Logic bypass method for controlling register accesses.	142
7.6	Control circuit for the first stage of the ACS.	144
7.7	Control circuit for the second stage of the ACS.	146
7.8	Refetch control for the second stage of the ACS.	147
7.9	General structure of the low-latency FIFO used in the order unit.	149
7.10	A tag cell used in the register scoreboard.	150
7.11	General architecture of the PC unit.	151
7.12	Interface circuitry for the DC and ACS write back phases to the PC.	152
7.13	Control schema for the PC unit.	153
7.14	Register cell used for scan testing the outputs from each stage.	156
7.15	A microphotograph of the <i>ECSTAC</i> microprocessor.	163

8.1	General structure of the <i>ECSCCESS</i> microprocessor.	173
8.2	General structure of the globe.	174
8.3	Control structure for governing RAW hazards.	175
8.4	Control structure for governing WAR hazards.	176
8.5	Control structure for governing the operation of the FU.	177
8.6	Generating a return event to the sun from 32 FUs.	178
8.7	Overall control structure of the combined sun and moons system.	180
8.8	General structure of the globe controller.	180
8.9	General structure of the PC controller.	181
8.10	General structure of the branch unit.	182
8.11	General structure of the stack unit.	183
8.12	A driver component used in the switching network.	186
8.13	Floorplans for <i>ECSCCESS</i> based on two different widths of a FU.	187
8.14	Block diagram for processing interrupts.	192

List of Tables

1.1	Dual-rail encoding of a binary value.	10
3.1	Relative gate delays in ECS assuming similarly sized transistors and loading.	29
3.2	Representations of the <i>agate</i> , <i>send</i> , and <i>feed</i> gates in various other paradigms.	31
3.3	Representations of the <i>agate</i> , <i>send</i> , and <i>feed</i> gates using Martin's CSP. . . .	31
3.4	Order of precedence for the ECS operators.	45
3.5	Various properties of the ECS operators.	45
4.1	Relative speeds of three gated pulse circuits.	54
4.2	Relative speeds of five implementations of the data latching circuit.	63
4.3	Relative speeds of three implementations of a conditional triggering circuit.	66
4.4	Relative speeds of two implementations of a generalized conditional triggering circuit.	67
5.1	Relative performance of six FIFO circuits.	79
5.2	Relative performance of four delay modelled pipeline circuits.	82
5.3	Three different design paradigms for precharge pipelines.	85
5.4	Comparison of precharge pipelines implemented with a decoupled four phase controller.	92
5.5	Comparison of precharge pipelines implemented using ECS.	95
5.6	Comparison of the ECS and decoupled four phase precharge pipelines assuming 1ns computation, precharge, and detection delays.	97
6.1	Conditions for generating the output carry of a full adder.	98
6.2	Three states required for implementing self-timed logic.	100
6.3	State encoding of dual rail carry propagation signals.	102
6.4	State table for the dual rail carry propagation signals.	103

6.5	Comparison of three self-timed adders.	106
6.6	Comparison of three pseudo self-timed adders.	110
6.7	State encoding of the AMULET carry propagation signals.	111
6.8	Comparison of ECS and AMULET adders, detection mechanisms, and a known fast adder.	112
6.9	Simulation results of a self-timed incrementer.	114
6.10	Simulation results of a 32 bit comparator tree.	119
6.11	Radix 4 Booth's algorithm for encoding partial products.	122
6.12	Combinations of two partial products which produce an output sum. . . .	124
6.13	Recoding of Booth's algorithm to provide the partial products.	125
6.14	Simulation results and comparisons of a 32 bit signed integer self-timed multiplier.	127
7.1	Register accessing requirements of the fundamental instruction set.	140
7.2	Statistical information from Hspice simulations of each <i>ECSTAC</i> sub-system.	157
7.3	Relative instruction frequencies used for the general testing of <i>ECSTAC</i> . .	159
7.4	Simulation speeds of <i>ECSTAC</i> for unit specific instruction streams.	159
7.5	Power estimations of each unit for a typical instruction stream.	161
7.6	Simulation speeds of <i>ECSTAC</i> for <i>bottlenecked</i> instruction streams.	161
7.7	Comparison of performance characteristics of various asynchronous micro- processors.	164
8.1	Instruction frequencies used in generating code for <i>ECSCCESS</i>	189
8.2	Simulation speeds of <i>ECSCCESS</i> for varying DC times.	189
8.3	Speed comparisons of various superscalar asynchronous microprocessors. .	191
A.1	Fundamental temporal equations and their corresponding gates.	201
B.1	ALU instructions.	205
B.2	Branch instructions.	206
B.3	Special instructions.	207
C.1	Instruction formats.	208
C.2	Fundamental instruction set.	208

C.3	Interrupt instructions.	209
C.4	32 bit constants for the LDC instruction.	210
C.5	Proposed FU allocations for <i>ECSCCESS</i>	212
C.6	Arithmetic instructions.	213
C.7	Multiply, divide, and sqrt instructions.	213
C.8	Logical and shifting instructions.	213
C.9	Comparator instructions.	214



Chapter 1

Introduction

OVER the last decade asynchronous systems have received a resurgence of interest amongst the scientific community [MBL⁺89a, FDG⁺94, Kes95]. This has largely been spurred by the difficulties currently encountered in clocking large systems [DWA⁺92], issues which will only complicate further as feature sizes reduce, die sizes increase, and the use of multi-chip modules (MCMs) increases. Furthermore at the dawn of digital circuit design a synchronous approach was chosen since it avoided the race and deadlock hazards of asynchrony. However, automated techniques have since been developed which avoid these hazards entirely, and in fact can now provide greater reliability in their asynchronous synthesis [Mar86, vBS88].

Asynchronous systems also possess a number of properties which make them more advantageous for certain applications than their synchronous counterparts. However, given that there are numerous approaches to asynchronous design which are as different from each other as they are with synchronous approaches, the ability to utilize these advantages varies. Conversely, an asynchronous implementation also introduces difficulties which are avoided in a synchronous one, and again the degree of disadvantageous behaviour exhibited by each of the asynchronous paradigms can be significantly different.

It is therefore the focus of this chapter to discuss the relative merits of both asynchronous and clocked systems, as well as the relative merits of the many different paradigms which can be used in the design of asynchronous circuits.

1.1 Advantages of asynchronous systems

1.1.1 Global communication

In a fully asynchronous systems there is no global clock routing. Instead, the control signalling between blocks is effected by localized communication employing a form of request/acknowledge (handshaking) protocol. The task of routing these localized communication signals is significantly less arduous than that of global routing.

Furthermore, in routing a global signal one must be wary of the effects of clock skewing, which can cause some parts of a system to latch newly arriving data rather than the old data from the last clock phase (Fig.1.1a). For large systems this phenomenon requires special attention [DWA⁺92]. In an asynchronous system however the signalling protocol ensures that no part of a system begins computing new data until all subsequent stages which use that data have latched it (Fig.1.1b).

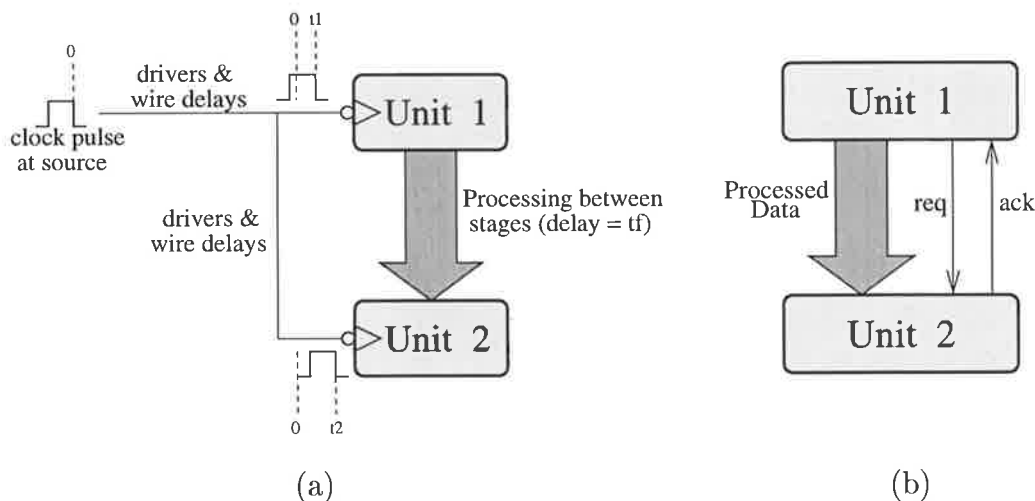


Figure 1.1: In (a) to prevent system failure the delay between t_2 and t_1 must be less than the fastest processing time t_f . In (b) Unit 1 cannot process new data (issue *req*) until Unit 2 has latched the data from the previous phase (issued *ack*).

Global clock signals are also required to drive a *large* number of gates, and to produce fast clock edges at these gates requires many driver stages which occupy a lot of silicon area ([DEC88] devotes approximately 30% of its die to this task). Cascading these driver stages to different parts of the system also complicates the skewing problem. Removing the global clock then potentially reduces the die size of a chip (however this must be countered by the silicon increase in providing the additional control for handshaking).

1.1.2 Data dependent computation times

Some functional units (such as adders, comparators, and incrementers) employ some degree of signal propagation in computing their result, and as such the computation time of that unit is dependent on the values of its source operands. For example, an incrementer with random input data will have no carry propagation 50% of the time (for data with an LSB of “0”), a one bit propagation 25% of the time (for data with an LSB of “01”), and so on, resulting in an average propagation length approaching only 1 bit as the data width approaches infinity. This is significantly shorter than the worst case propagation delay which is *equal* to the data width!

Self-timed computational units can therefore utilize this property in their architecture and provide a completion signal (such as *req* in Fig.1.1b) as soon as the computation is complete. Synchronous systems cannot utilize this property as the computation time for any one stage is fixed by the global clock period. Furthermore, this period *must* be long enough to account for the longest possible delay of the *slowest* stage of the entire system, which could be significantly longer than is necessary elsewhere.

Note however that in these instances it is the overall *latency* of the computation which is improved, not the *throughput*. This is because in terms of throughput, the rippling computation can be pipelined down to an arbitrary propagation length, to the extent that a synchronous implementation can out-perform an asynchronous one (due to the handshaking control overhead of the latter). However in terms of latency, the worst case scenario must still be accommodated by the synchronous system, which is only made worse by pipelining because of the larger number of stages required. This is especially so for large data widths, such as the 512 bit word size used in cryptography. By reducing the latency of computation through self-timed asynchrony, processor delays such as data hazards, branch target computations and program counter incrementing can all be improved.

1.1.3 Resilience to operating conditions

As a further extension to data dependent operations, self-timed asynchronous computations are also more resilient to operating conditions than are their synchronous equivalents. Factors such as temperature, supply voltage, and process spread all serve to alter the computation times of the same operation between chips (or even between different

locations on the same chip), so that the clock period must have an additional safety margin to encompass the vast majority of deviations. Those outside of this will not meet specification, and those within it will not be running at their maximum speed.

With self-timed logic the completion signal from any one stage is, like the computation itself, dependent on the operating conditions. Therefore a slower operating point will not cause failure, and a faster operating point will, in contrast to the synchronous case, improve the speed of the chip.

1.1.4 Reduced power dissipation

Power reduction in an asynchronous system results from the fact that inactive units do not generate any logic transitions and therefore have no power dissipation due to switching currents. In comparison, a synchronous system will have a significant portion of its power dissipated by the *continually* active clock pulses which drive a very large capacitance. Techniques such as clock gating can reduce this considerably, however it is infeasible to clock gate every hierarchical element of the design (due to the increase in area, skew, and complexity) to approach that achieved implicitly with asynchrony. With the use of dynamic logic, the removal of unnecessary precharge and activation phases can also further reduce power dissipation in the asynchronous case [FES94].

1.1.5 Incremental improvements

The fact that a synchronous chip must supply a clock period at least as great as its slowest component means that a new, faster design for some other part is redundant (in terms of clock speed) unless the slowest component(s) can also be improved (which may well require a complete re-design of the chip). Introducing a faster component into an asynchronous system however can in fact produce an improvement in the overall speed, since this is not purely governed by the slowest module as in a clocked design.

1.1.6 Synthesis and verification

Control circuits for asynchronous systems can be readily specified in a formal notation, such as transition equations [Puc90, Appendix 2], signal transition graphs (STGs)

[Chu87a] and trace theory [Ebe89]. Furthermore, VLSI programming languages for asynchronism such as Tangram [vBKR⁺91] and CSP [Mar90] have recently emerged as a means for an even higher level of design abstraction. These formalisms enable the rapid, automated synthesis of asynchronous control circuits which can also be formally verified to prove their correctness. Consequently, asynchronous systems can also be more reliable.

1.1.7 Power spectrum

Synchronous systems have a power spectrum dominated by multiples of the clock frequency, whereas asynchronous systems approximate a *white noise* spectrum. This is especially advantageous for microwave applications, where the integration of logic onto an antenna is required to have a negligible effect on the antenna characteristics. Furthermore, a self-timed asynchronous system is less susceptible to low-level electro-magnetic interference (EMI), because the propagation delays introduced by this will only slow down an asynchronous system [LZB92], but can cause failure in a synchronous one [CZ94].

1.2 Disadvantages of asynchronous systems

1.2.1 Control complexity

Although the problems associated with global clock routing are removed in an asynchronous system, other problems associated with the localized communication strategy are introduced. In particular, each individual stage of the design must have its associated control schema explicitly designed and tested to ensure its correct operation (this is in addition to the testing of the data path). Furthermore, each hierarchical composition of these stages must also be thoroughly tested. A synchronous system requires only one equivalent condition (in theory): that the clock period be greater than the stage delay.

1.2.2 Testability

If a synchronous design doesn't work because the clock frequency is too high then it's a simple matter to reduce this *post-fabrication* to achieve correct functionality. This cannot be done with an asynchronous system since no global control signal exists (techniques for mimicking this approach however are discussed in Section 7.5.1).

Asynchronous circuits are, by virtue of their greater number of interacting control signals, more susceptible to races, deadlock, and metastability than are synchronous circuits. Therefore the incorporation of testability is not only a more arduous task, it may also excessively reduce system speed, increase area, and may even *create* additional circuit hazards. Furthermore, many commercial test approaches rely on the fact that a finite number of clock pulses will occur between data inputs and data outputs, whereas in a self-timed system this is an unknown quantity, and can be especially problematic when integrated into a synchronous environment [WPS95].

The use of partial scan testing has reportedly been used with success in some applications of asynchronous logic already [HBB95, KB95, Ron94], as have other aspects of asynchronous gate-level testing [BR95, Haz92]. However the use of built-in self-test (BIST) techniques, which are used extensively in synchronous systems, remain untouched in the realm of asynchrony.

1.2.3 Area overhead

Although asynchronous implementations reduce silicon area by the removal of clock drivers, some asynchronous paradigms (in particular, dual rail logic) require approximately *twice* as many gates to be placed in the data path (incorporating both data *and* timing information with each bit). This is a considerable increase and a serious detriment to this particular paradigm. Furthermore, for small systems the area saved by removing the clock drivers (which in such cases may be minimal) could be overridden by the area introduced by the handshaking control.

1.2.4 Operating speed

In an asynchronous system there is a communications overhead which is required to implement handshaking between stages. This overhead can be quite excessive (in terms of the number of gates in the critical path) for a system with complex interconnectivity, and in many instances is a limiting factor to the overall system speed. A synchronous system however is only limited by the data processing delay of its slowest stage which, by careful design, can be made faster than the handshaking overhead of an otherwise asynchronous implementation.

1.2.5 Integration and software support

Another factor that discourages the use of asynchronous logic is that the market is overwhelmingly dominated by synchronous systems. Therefore integrating an asynchronous chip into such an environment requires the additional overhead of asynchronous to synchronous interfacing [CZ94, AML95b]. There is also a plethora of software programs geared towards easing the design of synchronous systems, and very little targeted for asynchrony.

1.3 Asynchronous paradigms

The pro's and con's of asynchrony outlined above give an indication of its more important properties, and if the advantages can be utilized without significant detriment from the disadvantages, then the topic of asynchrony becomes worthy of pursuit.

To this end it is imperative to investigate the range of asynchronous control schemas, signalling protocols, and operating modes which are available, particularly since not all paradigms share the same degree of advantageous (and disadvantageous) behaviour.

1.3.1 Timing assumptions

There is essentially a choice of three timing-related models which may be applied when designing asynchronous systems. A *delay-insensitive* (DI) model assumes that both the gate delays and the wiring delays are unbounded. This model is the most robust as it implies correct functionality regardless of both the place and route schema used and the drive strength of the gates themselves. However it is also the most expensive in terms of area and communication overhead, and in practice cannot be used without compromising the assumption of wire delays. In particular, the isochronic fork [Mar86] is often tolerated (albeit in contradiction to the DI assumption) as shown in Fig.1.2. Here, the delay between nodes A and B is assumed equal (but still unbounded) to the delay from node A to C .

A DI model with an isochronic fork is often dubbed quasi-delay-insensitive (QDI). However, this is in essence nothing more than a restrictive class of the *speed independent* (SI) model, whereby gate delays are still assumed to be unquantifiable, but the wiring

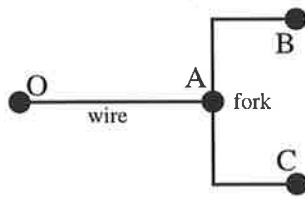


Figure 1.2: The isochronic fork.

delays are assumed to be zero. The SI model is better suited to practical circuit design than the DI model since a wider range of gates can be employed.

Finally, the least restrictive model assumes that the gate delays have an upper bound and is dubbed the *bounded delay* (BD) model. This is the assumption used in synchronous designs. It is less robust than the other two models and is therefore generally unsuitable for methods of formal synthesis and verification. In practice however it is the most realistic approach to use, since gate delays can in almost *every* circumstance be reliably quantified. As is evidenced by the *vast* array of synchronous chips, the BD model can be employed with relative ease to design functional, reliable circuits.

There are also two *modes* of operation which may be used with these timing models which specify the restrictions on the environment (that which supplies the stimulus to the design). The *fundamental* (FM) mode of operation requires that after an input change no further inputs be applied to a system until all its internal signals have settled, whereas the *input-output* (IO) mode enforces no such restriction. A fundamental mode circuit is easier to synthesize because it's guaranteed to be in a *safe* state when each new input is applied (thereby simplifying hierarchical composition), whereas an IO component requires a careful evaluation of each individual instance to meet this criterion. The IO mode however provides for greater design flexibility by being less restrictive on the environment's input signalling.

1.3.2 Control signalling

There are three main methods of effecting the handshaking protocol of Fig.1.1b. The first approach is dubbed *four phase* (4P) signalling and requires two control wires (one for *req* and one for *ack*) operating as shown in Fig.1.3a.

The rising edge of *req* (Δreq) signifies to the receiver that its data is valid, and once the receiver has finished with this data it issues Δack , at which time the sender's data

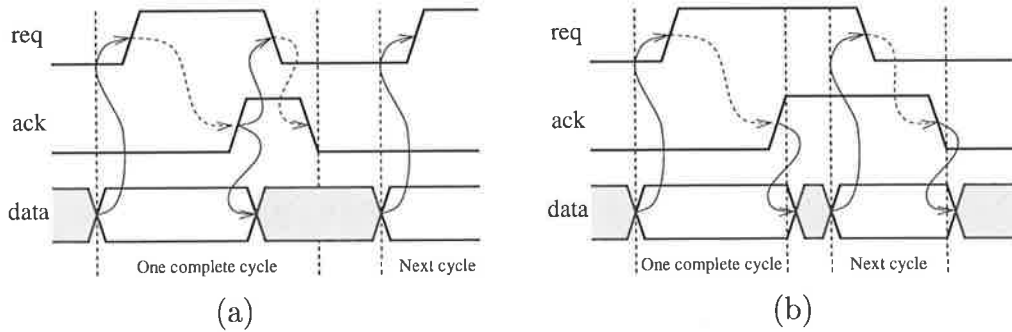


Figure 1.3: (a) A four phase handshaking protocol, and (b) a two phase protocol. Solid arrows indicate the sender's actions, and dashed arrows indicate the receiver's.

may become invalidated. To then complete the 4P signalling, the sender issues a falling edge on req (∇req) which is then followed by the receiver issuing ∇ack . Similar schemes exist in which the data validity cycle is spread across different phases of the 4P signalling (for example, from Δreq to ∇ack).

A *two phase* (2P) signalling protocol as shown in Fig.1.3b also requires two wires, and is identical to the 4P protocol until the receiver issues Δack . In contrast to the 4P protocol, this then completes the handshaking cycle, and a new cycle is initiated when ∇req occurs. Consequently the signalling direction is irrelevant in a 2P environment, and only one data validity scheme is possible. Although in one sense a 2P protocol can achieve a higher speed than a 4P protocol by removing the redundant return-to-zero (RTZ) phase, the control circuits necessary to implement the handshaking are more complex and therefore slower. In [DW95] it was shown that, in a speed independent model, 4P pipeline circuits actually run faster than 2P circuits.

Finally, in [vBB96] a method was proposed for executing a 2P handshaking protocol with only one wire, driven high by the sender and low by the receiver. This technique is relatively new and remains to be thoroughly investigated.

1.3.3 Signal encoding

Two different forms of signal encoding are predominant in asynchrony. The first is *dual rail* (DR) encoding, in which three data states are encoded onto two wires as in Table 1.1. This approach enables timing (validity) to be coupled with each bit of data and therefore allows accurate self-timed computations to be devised. It guarantees that a completion signal can only occur from a unit after the data is valid: a property that is often exploited in SI and DI synthesis. However, a doubling of the data width (which results from 2 wires

being needed per bit) and an increase in gate complexity means that approximately twice the area of a synchronous implementation is used.

Wires		State encoded
w0	w1	
1	1	not valid
1	0	logic 1
0	1	logic 0
0	0	error

Table 1.1: Dual-rail encoding of a binary value.

Single rail (SR) encoding is used almost universally in synchronous systems, in which one wire is used for each binary data value and timing information is decoupled. In an asynchronous system, this timing is then represented by additional handshaking signals to which the data information is *bundled*. This approach is less suited to formal methods as it can only approximate data validity (and therefore prevents self-timing), however its area usage is significantly less than with dual rail encoding.

1.3.4 Summary

Given the range of asynchronous implementations available it's not surprising that different styles are used for different purposes. For example, SI and DI models are popular amongst formalists because of their mathematical provability and are also commonly used in the design of complex systems (where automation from a set of rules is time saving). Single rail encoding is often used in the bulk of designs except where aspects of self-timing are to be exploited, when dual rail encoding is used (note that a conversion and area penalty is sometimes then suffered). BD models are often used in *engineered* designs (or as a form of post-processing to a synthesized SI circuit), for which gate level optimization techniques can be incorporated to increase system performance.

1.4 Thesis outline

This chapter has presented an overview of the merits of asynchrony, as well as introducing the many different design methods which can be used to implement asynchronous systems. The following chapter will present a summary of the most important contributions to

asynchrony, focussing on their paradigms, goals, results, and significance. The focus of the author's event controlled systems (ECS) methodology will also be outlined.

Chapter 3 will present an asynchronous library of gates and different approaches to their representation. The ECS representation developed by the author, which is intended to simplify and properly reflect asynchronous circuit behaviour, will also be introduced. Aspects of error detection based on this representation will also be given.

Chapter 4 will present a number of asynchronous design techniques which should be used in the design of high speed circuits, and chapter 5 will present some very fast pipelining circuits for use with both static and dynamic logic. Chapter 6 will introduce some novel self-timed design techniques and architectures as well as a new single rail approach to self-timing, dubbed *pseudo self-timing* (PST).

Chapter 7 will detail the design of a pipelined asynchronous processor called *EC-STAC*, including its ISA, high level implementation, control structures, test strategies, and simulation and fabrication results. Chapter 8 will likewise detail the implementation of another processor *ECSCCESS*, aimed at increasing parallelism and fully utilizing PST techniques. Both of these chapters will compare the performance of these ECS processors against others previously reported.

Chapter 9 will then present conclusions from the work presented in this thesis as well as some ideas on how it can be extended for future applications.

Chapter 2

Related Work

THERE has been a vast amount of research on asynchronous systems design, and to review every aspect of this discipline is impractical and unnecessary, since excellent comprehensive reviews can be found in [Pee, MU, Hau95, GJ90]. Only a summary of the most important contributions which have been made to the field is therefore presented in this chapter, with an emphasis on those aspects of asynchrony which have been tackled by various research groups, and why their results have been significant.

From such a review it should be possible to determine those areas of asynchronous systems design which have proved inferior when compared to clocked systems. This will then indicate directions for further investigation from which the focus of this thesis will be drawn.

2.1 Synthesis and verification

The applicability of asynchrony to formal methods has spawned a vast network of research groups who have endeavoured to harness this property into algorithms for both synthesis and verification. Synthesis begins with a high-level system specification and reduces this through various processes into a circuit-level implementation (this also incorporates aspects of verification at the high-level). Verification often begins with a low-level specification of a system (usually incorporating a SI or DI model) and then determines the locations of potential hazards.

2.1.1 Tangram

Tangram [vBKR⁺91, vBR95] is a high-level programming language based on Hoare's communicating sequential processes (CSP) [Hoa85] and Dijkstra's guarded command language [Dij76]. A specification in Tangram is similar to a conventional programming model, enabling procedures, loops, conditional execution, sequentiality and parallelism, etc.

A Tangram program is translated into a set of *handshake* components which provide an intermediary system description prior to gate-level synthesis [vB93]. Handshake components communicate along *localized* request and acknowledge channels initiated by active and passive elements respectively. Some examples of handshake elements include a JOIN (to synchronize two or more concurrent processes), a SEQ (to enable signal transfers between passive and active components), and a VAR (to store the value of a variable).

Compilers have been devised by Philips which translate these handshake circuits into different asynchronous targets, including both four and two phase single and dual rail circuits. Their design approach provides for a very fast and reliable compilation of asynchronous circuits, and they have used it to design a number of variations of asynchronous error detector chips for digital compact cassette and CD players (see Section 2.2.3). Their work has shown conclusively that an asynchronous circuit can dissipate significantly less power than a synchronous equivalent, however their automated approach tends to result in a large number of gates in the control path, and is therefore unlikely to be of benefit in the design of high speed circuits.

2.1.2 Communicating processes

Alain Martin [Mar86] has also devised a high-level programming language similarly based on the work of Hoare and Dijkstra. It also translates the original specification into a series of handshaking elements and then decomposes these (using techniques akin to trace theory [Dil89]) into simple conditions which can be used to construct the final gate-level circuit. The target paradigm is however restricted to a 4P QDI implementation, which demands a large number of gates in the control path and is therefore unlikely to be of benefit for high speed circuit design. A number of systems, including an asynchronous microprocessor, have been fabricated using this synthesis technique (see Section 2.2.4).

2.1.3 Signal transition graphs (STGs)

STGs are interpreted Petri Nets whereby places become nodes, transitions become arcs linking the nodes, and the number of tokens in a place is limited to one. They were first developed by Chu [Chu87b] and have since gained considerable popularity amongst the asynchronous community [PG93, HC95, JPKJ95] as a verifiable specification for SI circuit synthesis.

As an example a STG specification of a Muller C element (or *cgate* for short) is shown in Fig.2.1a, where the symbols + and - indicate positive and negative transitions of the associated variable. The *cgate* is often dubbed the *AND* gate to events, as it only generates an output event when transitions on both of its inputs (in the same direction) have occurred. Techniques exist for verifying that a STG meets the required criterion for a decomposition into a hazard-free SI circuit [Chu87b].

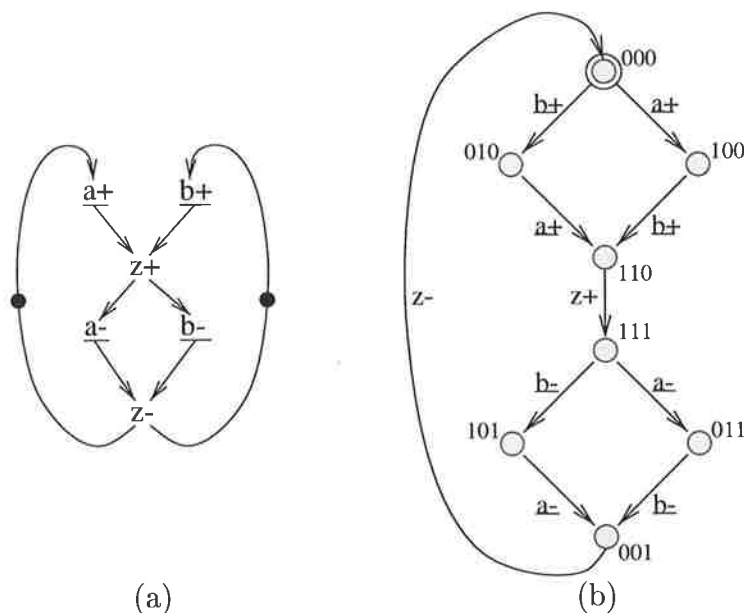


Figure 2.1: (a) A signal transition graph and (b) a corresponding state graph for the *cgate*, with inputs a and b and an output z .

A STG is first contracted into a subset of STGs for each variable, and these are then transformed into a state graph with unique state assignment (such as in Fig.2.1b) for synthesizing the logic function, which for the *cgate* is: $z = a.b + z(a + b)$.

2.1.4 STG related synthesis

Myers [MM93] has extended the STG synthesis routine to incorporate timing information onto each arc. This enables timing-redundant arcs to be removed from the STG which subsequently reduces the complexity (and increases the speed) of the synthesized circuits.

A group at the University of Berkeley have produced an extensive design environment called SIS [SSL⁺92] which incorporates the use of STGs as well as other specification methods. SIS provides an X-interface for STG based synthesis and incorporates a range of optional optimization programs to improve the final circuit's performance (through logic minimization and timing assumptions).

An independent extension to the graphical specification of STGs is called *Change Diagrams* (CDs) [KKT⁺92]. CDs enable an initial input phase of the graph to be specified prior to the *cyclic* phase required by STGs. Furthermore, the arcs between nodes are given additional properties which enable AND and OR transitions as well as the ability to remove pending events. These extensions improve upon the restrictiveness of a STG specification.

2.1.5 Other contributions

Josephs and Udding [JU90a] have devised an algebraic approach to the design of DI, SI, and handshake circuits which has been used to synthesize some simple systems (such as a *toggle* and a FIFO), however the algebra is rather cumbersome even for simple gate specifications. Ebergen [Ebe89] has provided a synthesis technique based on trace theory specifications. His approach has been used to generate 2P, dual rail circuits [EP92] which are extremely area intensive. For example, a simple half-adder cell requires over 190 transistors [Hau93] in its implementation.

2.2 Asynchronous hardware

The majority of asynchronous research groups have focussed their attention on the topics of synthesis and verification. Consequently there are relatively few examples of fabricated asynchronous chips, and those that have been done are mostly small, experimental design projects. However there have been a few significant milestones in this area.

2.2.1 Micropipelines and the CFPP

The micropipeline design style was first introduced by Sutherland [Sut89], and has since been used by a number of research groups as a basic architecture for pipelined processor implementations [KdSRA91, FDG⁺93, CL95]. Micropipelines use a 2P, speed independent, bundled data protocol to control the activity between adjacent stages (utilizing the *cgate*) and incorporate dedicated event controlled latches between stages (Fig.2.2a).

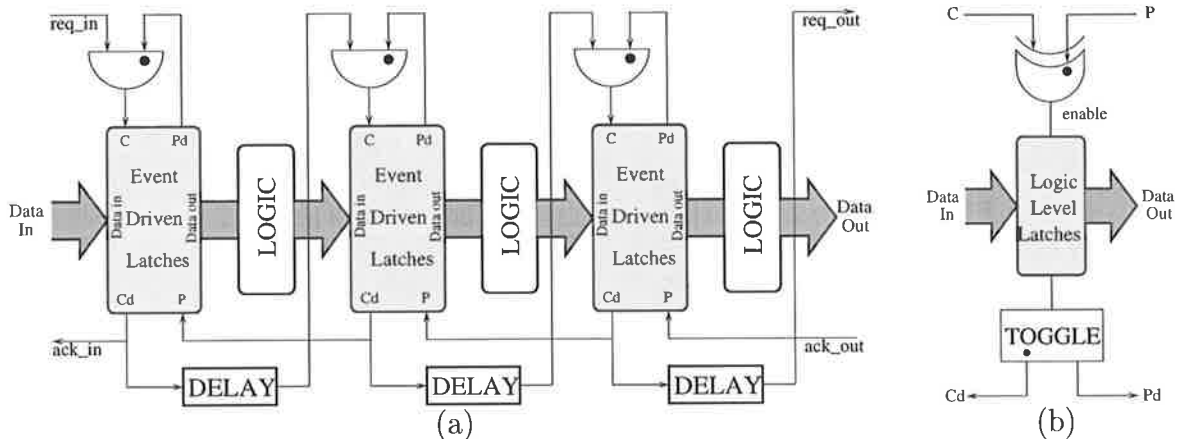


Figure 2.2: (a) A simple micropipeline architecture with delay-modelled processing between stages, and (b) an alternative controller for use with smaller, logic-driven latches.

A selection of macromodules exist which enable more complex designs to be constructed. Examples of these are a *select*, which steers input events to one of two output events depending on a governing control signal; a *call*, which enables two separate event streams to access a common unit; and a *toggle*, which alternately steers input events to its two outputs. As an example construction of these units, an alternative latch controller for the event driven latch is shown in Fig.2.2b. The *enable* signal of this design can be used to interface to conventional logic-driven latch designs, which are significantly smaller than the event-driven latch (for which various implementations are given in Fig.3.11).

An extension to the micropipeline is the counter-flow pipeline processor (CFPP) [SSM94], which is a micropipeline style architecture devoted to processor implementations. The fundamental difference is that the CFPP enables both the forward and reverse flow of information through its pipeline (for instructions and data results respectively), thus reducing the need to stall for register write-backs. An issue against the CFPP is that arbitration between flows is needed at every stage (which increases the cycle time), and the data path is large (requiring a lot of latches per stage).

2.2.2 AMULET I and II

The AMULET group at Manchester University developed an asynchronous microprocessor [FDG⁺93, Pav94] based on the (almost entire) ISA of an already commercial processor, the ARM6. Their processor, AMULET1 utilized a 2P micropipeline style architecture and incorporated some interesting design techniques for register-locking and interrupt handling. In implementing the ARM6 asynchronously the group focussed on exploiting the properties of low power consumption.

Although the AMULET1 design failed to improve upon the synchronous ARM6 implementation in all aspects, it did demonstrate the feasibility of implementing complex systems asynchronously. Furthermore, this was the first asynchronous attempt at a sixth generation synchronous machine, and as such the fact that their performance (in most critical areas such as power and speed) came within a factor of two is encouraging.

The AMULET group have recently completed a 4P implementation of the ARM6 processor called AMULET2, which has provided almost double the speed of the 2P micropipeline approach (using a SI control model). Branch prediction and caching have also been incorporated, along with more refined low-power techniques, to improve performance.

2.2.3 DCC error detector

The group at Philips who devised the Tangram VLSI programming language have also used this to generate a number of asynchronous chips. In particular, an error detector chip for a digital compact cassette player [vBBK⁺94]. The group have targeted their designs for low power dissipation, which is convenient since the DCC chip has a very slow speed specification.

Since the first “first-time-right” chip was fabricated the designers have aimed at improving the area usage, testability, and portability of their approach. In particular, the initial dual rail design has been re-implemented using single rail data [vBBK⁺95], and the asynchronous cell library has been implemented using a *generic* gate library to facilitate technology remapping.

They have integrated this chip into a DCC player and have demonstrated its functionality. The SR chip consumes approximately 20% of the power of its synchronous equivalent with only a 20% area overhead.

2.2.4 Caltech microprocessor

The VLSI programming language developed at Caltech has been used to design a simple 16 bit asynchronous RISC-style microprocessor [MBL⁺89a]. The language is targeted primarily at producing a verifiably correct design, and the chip was reported to operate (at 5V supply) at 18 Mips drawing 45mA of current [MBL⁺89b]. Being a QDI design the chip exhibited a high tolerance to operating conditions such as temperature and voltage.

2.2.5 Other contributions

A self-timed floating-point divider has been implemented using a SI model in [WH91], which has also been incorporated into the commercial SPARC64 processor reported in [WPS95]. Current-sensing has been investigated by [DDH91, GJ95] as an alternative means of signifying completion detection in an asynchronous pipeline, and numerous groups have worked on the design of sub-systems such as buffers [CL86, YHJN95] and adders [Gar93, DA93]. Some work on processor design has also been done at the architectural level in [ECFS95, End95a], and the author reports on the implementation of an asynchronous FFT and a high speed microprocessor in [MAL94, MAL95].

2.3 Summary of related works

A lot of the work done in asynchrony has been in the area of synthesis and verification, and in particular using the SI and DI models. Furthermore the trend recently has shifted from 2P micropipeline style architectures to 4P pipeline control (which under such paradigms has been shown to generate smaller and faster circuits). There has also been a shift towards using single rail data with combinational logic between stages (as in a synchronous machine) rather than dual rail data, and then bundling this to SI control with the computation time between stages modelled as a lumped delay (as in Fig.2.2a). This approach compromises the SI control by assuming a BD model for the computational delay only.

The focus of most hardware groups has been on utilizing asynchrony for low power consumption, with system speed as a secondary concern. Aspects of testability and self-timing have also been of interest to these groups.

2.4 A need for speed

In considering the research groups who are involved with hardware design it is evident that asynchronous logic has in fact demonstrated improvements over synchronous designs in the area of power consumption. When specific efforts are made towards implementing low-power circuits, these improvements are substantial, however it seems also that power reductions can still be achieved even without a significant focus on this.

In contrast, asynchronous designs have not been shown to exhibit any marked speed improvements over synchronous designs (except for a few rare instances). In fact, the converse is generally true. The bulk of asynchronous hardware has been speed limited by the control overhead and has therefore exhibited a notably slower speed than could be implemented synchronously. This issue is a severe impediment to the adoption of asynchrony, since processor speed is often a more important issue than power consumption.

It is therefore of interest to devise techniques which enable faster asynchronous control circuits to be implemented. In this quest it is hoped that the speed deficit of asynchrony will be reduced, perhaps even to the point in which the speed surpasses a corresponding synchronous implementation for certain architectures. Furthermore, it is anticipated that the additional benefits of low power consumption will still occur in an asynchronous design even without explicitly designing for it. This then is the focus of this thesis: to devise fast asynchronous control systems which, if needs be, compromise other issues such as power dissipation.

The question which must then be asked is which of the asynchronous paradigms and design approaches presented thus far is most suited to the design of high-speed systems? To implement high speed circuits one must have a great deal of control over the low-level implementations. Consequently the synthesis techniques available are unsatisfactory, as they are restricted to a given set of rules and building blocks (such as handshake elements). The ideal behind synthesis is to *shelter* the designer from low-level design aspects and deal only with the high-level architecture (such as inter-connectivity and system functionality) and this is in direct contrast to the requirements for high speed circuits. Although post-optimization techniques exist for many synthesis approaches they still do not provide for complete control over the low-level implementation and are limited in their scope.

It is of course possible to further optimize synthesized circuits at the gate level. This

may involve incorporating dynamic logic, removing unnecessary gates based on timing assumptions, re-implementing complex structures etc., but it is still to be expected that the initial synthesis routine could be bettered by an *engineered* approach whereby all aspects of the design process are controllable and more implementation options are available at all stages of design.

It seems necessary then to abandon formal synthesis techniques and concentrate on devising engineered control circuits, since these provide for greater design flexibility. Consequently the asynchronous environment considered by the author will in general be a bounded delay system operating in the IO mode, as this is the least restrictive model available.

A single rail encoding of the data stream will also be used for a number of reasons. Firstly, the gate complexity of single rail circuits is simpler than for dual rail which should therefore result in faster circuit speed, and the area usage is less which therefore reduces cost. Furthermore, the familiar gate structures used in clocked designs may be employed which facilitates the use of current standard and generic cell libraries.

The last issue to resolve then is whether or not to implement 4P or 2P control circuits. Now although 4P has been demonstrated to be faster than 2P for speed independent circuits, the same is not necessarily true if bounded delay assumptions are used. This is because low-level engineered circuit design has not been extensively applied to both 4P and 2P approaches to enable an accurate comparison. It has been quoted that a lot of the speed reduction in 2P is due to the greater gate complexity, however in a BD model many of these gates can be *non-acknowledged* (given timing assumptions) and can therefore be removed from the critical path. The same can be said of 4P of course, however the *additional* RTZ phase must also be accounted for which complicates the design process and reduces the number of gates which can be non-acknowledged. The 2P design paradigm is therefore used which, as will be shown, can result in faster circuit implementations than is possible in a 4P paradigm.

To summarize then, the design environment considered by the author to facilitate high-speed circuits is a two phase, single rail, bounded delay, bundled data, input-output model.

Chapter 3

Event Controlled Systems (ECS)

Design Representation

FOR engineering asynchronous control systems one must have both a *library* of basic elements for circuit composition as well as a means of representing the functions of these elements for verification. Now although these elements can be represented in terms of binary logic equations (as per the *cgate* of Fig.2.1), this may not be the best representation to use for simplicity and descriptiveness.

This chapter will first outline the basic set of event controlled gates used by the author in the composition of asynchronous circuits, of which many are synonymous to the basic elements of micropipelines [Sut89] but with some significant exceptions. These exceptions arise from a new gate representation which is explained in Section 3.5 and which is considered by the author to best reflect their functionality. Techniques for circuit verification and error detection will then be presented.

3.1 Conventions

In the gate descriptions to follow a number of conventions have been adopted. Firstly, a distinction is made between event lines, which are used to effect the 2P communication protocol and for which only the logic *transitions* are relevant, and data lines, which transmit conventional binary information and whose logical *state* is of relevance. Consequently, the event lines are drawn diagrammatically with arrowheads and data lines without. Similarly, the name of an event line is prefixed with either a “*d*” or a “*o*”

symbol, to differentiate them from data lines.

Secondly, primed events are indicated on the gate symbols by a filled dot, and in a temporal specification (described in Section 3.5) with an overline. These are events which are *assumed* to have occurred at initialization and therefore set the initial state of that gate. A primed event equates to an inversion of the signal at the gate input, although in some instances an alternative gate implementation is used.

3.2 Event controlled elements

The range of basic event controlled gates used in the composition of asynchronous circuits is described below. To further facilitate their understanding, each gate description is accompanied by a timing diagram, symbol, and one or more VLSI schematics or circuit diagrams. Relevant variations from the micropipeline gate library are also discussed.

It is important to note that a number of gates have the same logical functionality but are treated differently due to their conceptual differences in the ECS framework. Consequently, some gate symbols also differ from those in the corresponding micropipeline library. This issue is further compounded by the fact that 2P event control wires (called *event lines*) and binary data signals (called *data lines*) are conceptually different in the ECS framework.

3.2.1 Muller-C element

This is often known simply as the *cgate*, or *rendezvous*. Its logical function is to transfer the state of its inputs to the output when all of these are at the same state (as indicated in Fig.3.1).

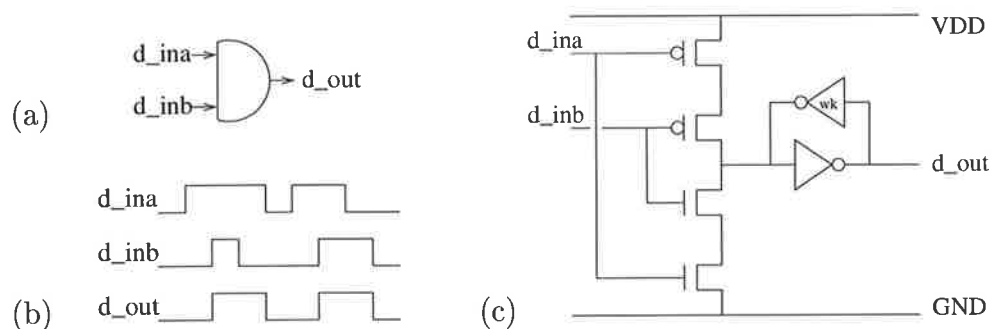


Figure 3.1: (a) Circuit symbol, (b) timing diagram, and (c) static VLSI schematic of the *cgate*.

The *cgate*'s behaviour is however better described in terms of its transitions (which is also more relevant to the 2P control schema being employed). In this respect, a transition on its output ∂_{out} is produced once transitions have occurred (in the same direction) on all of its inputs. For this reason the *cgate* is often considered to be the *AND* function for events, hence the similarity in the circuit symbol. Note that in the ECS design framework discussed in Section 3.5, two successive events on an input event line are prohibited unless an output event occurs between them. An n -input *cgate* can be constructed by extending the n and p transistor trees in Fig.3.1c, or by a combination of these basic 2-input elements.

3.2.2 Merge gate

The *merge gate* performs the *OR* function for events, as a transition on either of its inputs is translated into a transition on its output. Note however that it is unrealistic (in terms of the VLSI implementation) for two input transitions in close proximity (one on each event line) to result in two transitions on the output, as propagation delays and rise and fall times prevent this from happening. This scenario must therefore be prevented, and input transitions must be adequately time-separated so they can be processed individually by the subsequent circuitry (Section 3.5.8 provides a method for detecting when this requirement is violated).

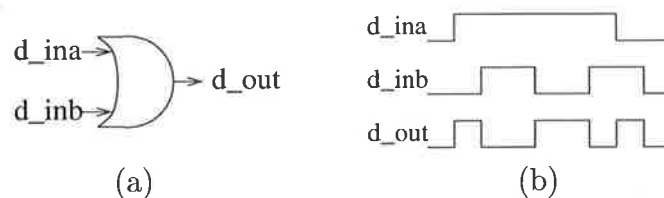


Figure 3.2: (a) Circuit symbol, and (b) timing diagram of the *merge gate*.

It is evident from the timing diagram in Fig.3.2b that the *merge gate* is nothing more than an *xor* in binary logic, and an n -input *merge gate* can be constructed from a combination of these basic 2-input gates. Two VLSI implementations of an *xor* are shown in Fig.3.3.

The 6 transistor design is the most compact of the two but does not have a very good drive capability, so for driving high loads a similarly structured *xnor* is often used with an inverter buffer on the output. The 8 transistor design is faster and can drive higher

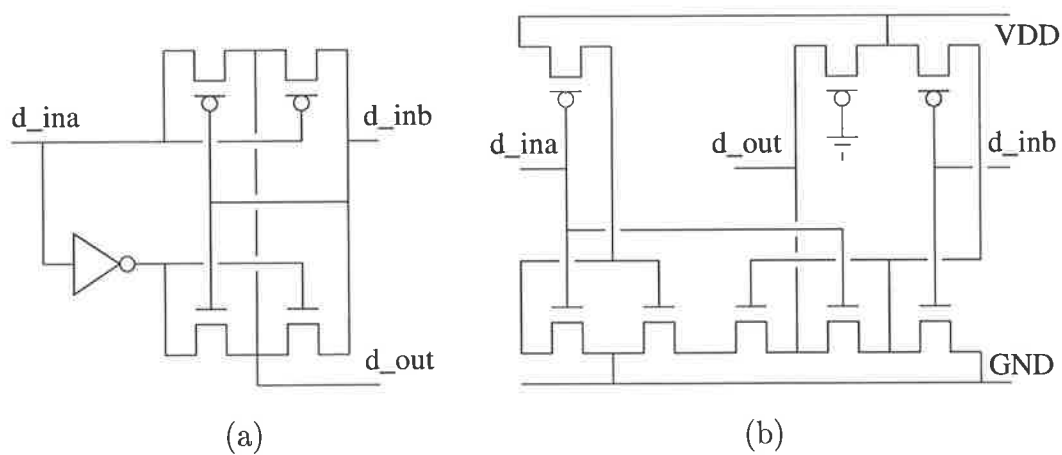


Figure 3.3: (a) 6 transistor design, and (b) fast pseudo-nmos 8 transistor design of an *xor* gate.

loads than its 6 transistor counterpart, but its noise margin is worse and it dissipates more power. It is only used in critical event paths or where trees of *xors* are required (such as in parity detection or the merging of numerous events), for which the sequential pass transistor paths of the 6 transistor design produce excessive delays. Conventional multiplexer based *xors* are also used on rare occasions, and for some control circuits the fast and small 6 transistor *xor* of [WFF94] is used (although this was only discovered after the majority of designs were already completed).

3.2.3 Send gate

This gate passes an input event to the output when a controlling data signal is high. Furthermore, an input event which occurs when the controlling signal is low will be kept pending until the control eventually does go high, at which time the pending event will be propagated through to the output. Fig.3.4b illustrates the *send* gate's operation.

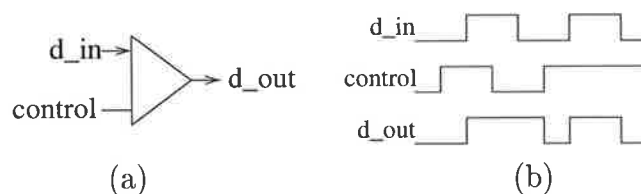


Figure 3.4: (a) Circuit symbol, and (b) timing diagram of the *send* gate.

This gate is not to be found in the micropipeline library although its implementation is identical to that of a logic-driven latch. The distinction is made however since the send gate operates with event lines whereas a latch operates only on data. Also, the send

gate must be initialized to force the output into a known state. Two VLSI schematics of a *send* gate (or latch) are shown in Fig.3.5, and initialization for an output low can be incorporated by inserting a pull-down (or a pull-up for Fig.3.5b) transistor at the node labeled *init*.

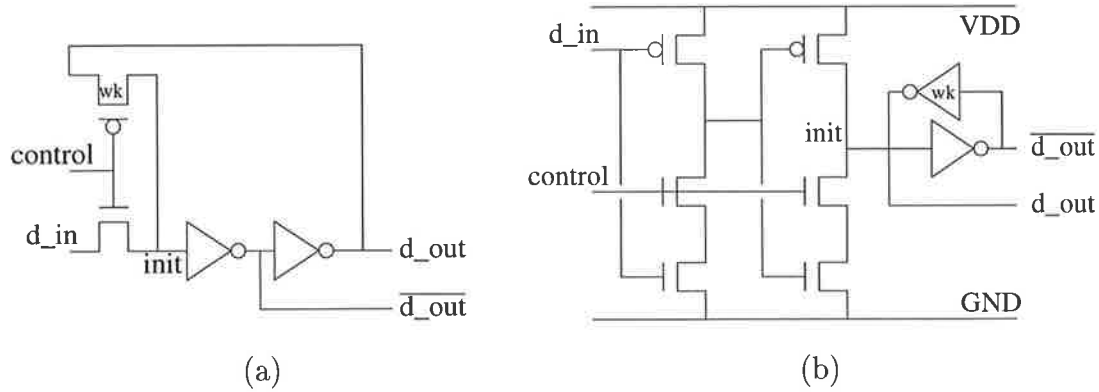


Figure 3.5: (a) Compact latch circuit and (b) a fast inverting Svensson latch design.

The circuit of Fig.3.5a is very compact and is therefore often used as data latches or for non-critical event paths. The circuit of Fig.3.5b [YS89] is a faster (but larger) design used in critical event and data paths.

The *send* gate provides a useful method for stalling an event until some other controlling condition becomes true, as well as enabling fast bypass techniques for conditional pipeline execution (see Section 7.4.3 for example). Note that in the ECS design framework explained in Section 3.5, successive input events are prohibited unless an output event occurs between them.

3.2.4 Feed gate

The *feed* gate is similar to the *send* gate in that it too passes an input event to the output when a controlling data signal is high, however in this instance an input event which occurs when the control is low is not kept pending, and is therefore *never* propagated through to the output. This fundamental difference can be seen by comparing the second event in the timing diagrams of Figs.3.4b and 3.6b. In the former, an output event is eventually generated (when *control* goes high) whereas in the latter no such event occurs.

By combining two *feed* gates (driven by *control* and its complement), the *select* module in the micropipeline library is created. Note however that in the design of the *feed* gate

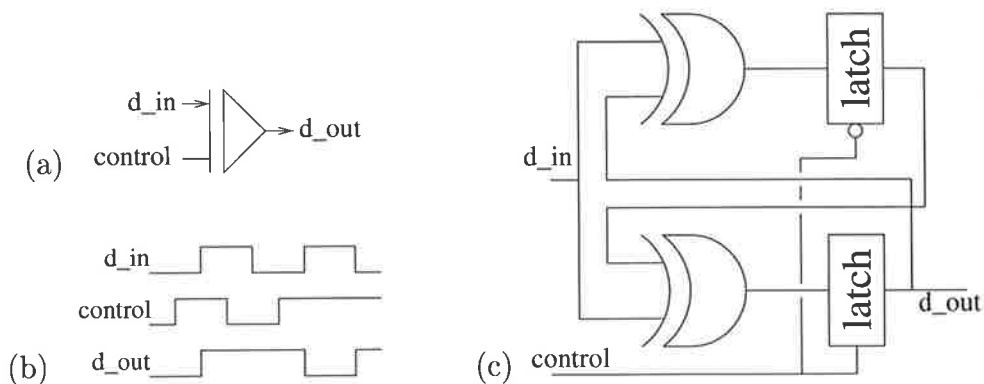


Figure 3.6: (a) Circuit symbol, (b) timing diagram, and (c) circuit diagram of the *feed* gate.

(shown in Fig.3.6c) the complement-driven event output is also available from the top latch, so that only one *feed* gate is actually needed in practice for its implementation (although conceptually two *feed* gates are still used). The symbol chosen for a *select* module is shown in Fig.3.7.

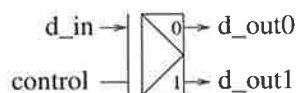


Figure 3.7: Circuit symbol of the *select* gate.

The *feed* gate is often used in practice as a *select* module as well as for conditionally triggering the operation of a sub-system.

3.2.5 Restore gate

The *restore* gate is similar in functionality to the send gate but with a controlling event instead of a controlling data line. That is to say, an event which is pending at the input to the *restore* gate will propagate through to the output when a *subsequent* event occurs on the control line d_{pass} , as shown in Fig.3.8. Note that if no input event is pending, then an event on the control line will have no effect on the output. For example, the second event on d_{pass} in Fig.3.8b.

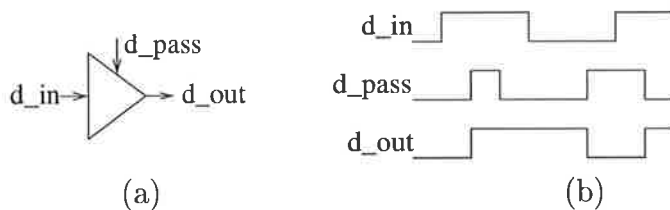


Figure 3.8: (a) Circuit symbol, and (b) timing diagram of the *restore* gate.

The *restore* gate has no synonymous module in the micropipeline library however it is similar in functionality to the *decision-wait* unit popular in other paradigms [Kel74, JU90b], whose more restrictive functionality can be replicated by using two *restore* gates. Note however that the *decision-wait* element is often used to construct a *call* module in the micropipeline library (which grants access of a sub-system to one of two non-conflicting input requests), of which the *restore* gate is therefore primitive.

Figure 3.9 shows two different circuit diagrams for the *restore* gate. The first design simply generates a short pulse (using a self-timed pulse circuit as described in Section 4.2.2) each time an event on $\partial pass$ occurs, and if an event on ∂in is pending, propagates this to ∂out via the *send* gate. This design is small in size but incurs a delay of 3 gates between $\partial pass$ and ∂out , and is therefore used in non-critical event paths.

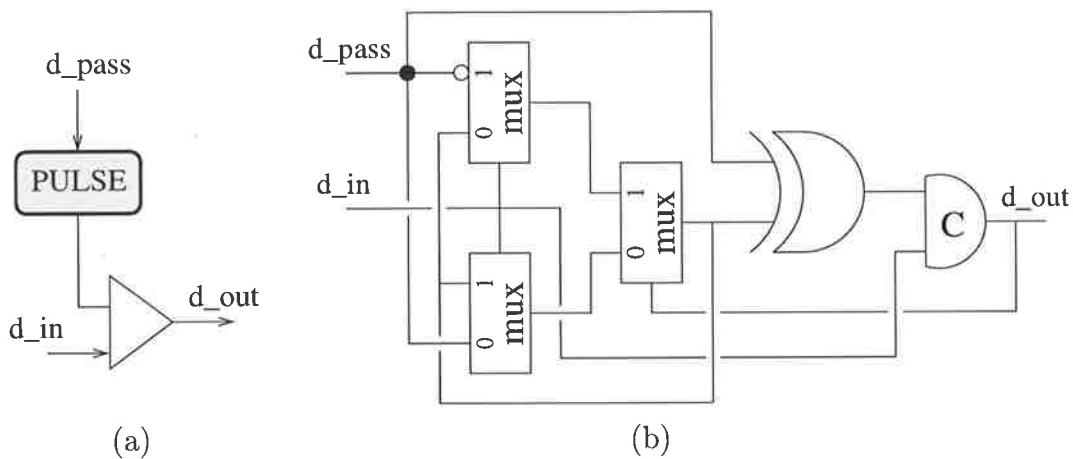


Figure 3.9: (a) Pulse driven circuit, and (b) sequential logic circuit for the *restore* gate.

The second design is derived from a state transition graph using asynchronous sequential logic techniques [Puc90, Chapter 6], and consequently views the events as binary signals. Note that a *C-element* is used in this implementation but that its functionality should be interpreted differently from that given in Section 3.2.1, since in this instance it's used as a logical block (obeying the logic equation given in Section 2.1.3) and not as an event controlled unit. This second implementation is larger than the first but incurs only a 2 gate delay between $\partial pass$ and ∂out , and is therefore used in critical event paths.

3.2.6 Until gate

The *until* gate can be crudely viewed as a two to four phase converter for event lines. It sets an output data signal high after an event on one input occurs, and then sets it low

again after an event on the other. It is quite obvious from this that the *until* gate can be implemented as an *xor* if the events on the two input lines are alternating (an enforced restriction), however it is viewed conceptually as a different gate because its inputs are event lines, not data lines.

It is useful to be able to ascertain from the circuit diagram which of the two input events to an *until* gate sets the output data high (or low, as the case may be), and also to see what the *initial* state of the data is prior to any events occurring. For this reason the circuit symbol of Fig.3.10a is used for the *until* gate, as opposed to the conventional *xor* symbol. The event which strikes the outer bar of the symbol sets the output low, and the primed event (signified by the dot) indicates the initial state of the output data.

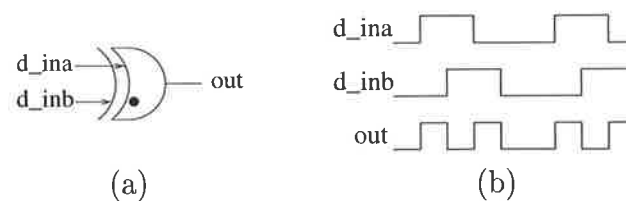


Figure 3.10: (a) Circuit symbol, and (b) timing diagram of the *until* gate.

Whether an *until* gate is implemented as an *xor* or *xnor* gate depends on the initial states of the inputs and the required initial state of the output, however if the convention is adopted whereby *all* event lines are initialized low, then the *until* gate shown in Fig.3.10a is implemented as an *xor* (if the dot were placed on the other input, then an *xnor* implementation would result). Note also that the first input event to occur must be on the *non-primed* event line (since they're constrained to alternate).

3.2.7 Latching element

Some event driven latching elements are proposed in the micropipeline library and can be implemented as shown in Fig.3.11. Both circuits assume the *hold* and *pass* events are initially low and are alternating, and that the latch is initially transparent. The problem with these designs is that they occupy a lot more silicon area than the logic driven latches shown in Fig.3.5, have a high load on the event lines, and also require the event line's complement to be generated. Logic driven latches are therefore used in almost every instance (except for when the data path is very small), and their circuit symbol can be seen in Fig.3.6d.

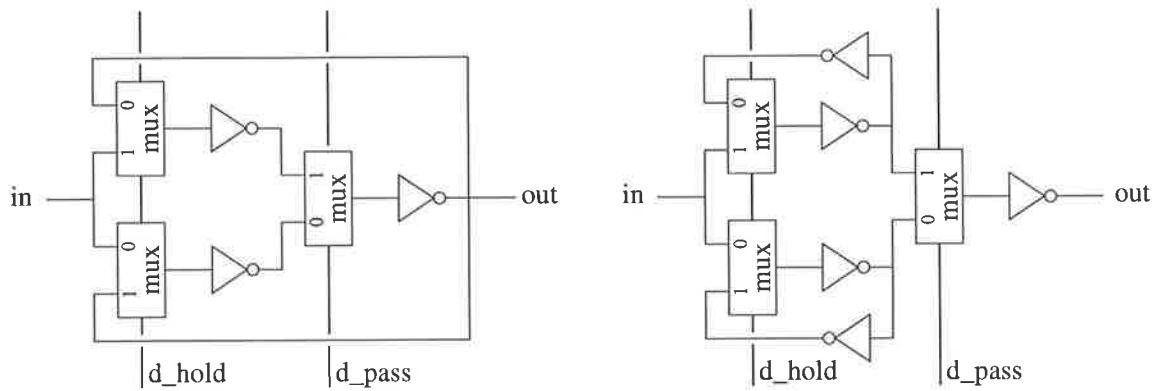


Figure 3.11: Two implementations of an event driven *latch*.

3.2.8 Delays and logic functions

It is often necessary (particularly in a bounded delay, IO paradigm) to insert lumped delays into an event stream, such as for the modelling of logic functions or to wait for some associated data to become valid. Delays are implemented simply as inverter chains and the symbol adopted for them was shown in Fig.2.2a.

Binary logic functions (such as *nand* and *nor*) are also used in devising control schemas but can only be applied to data lines.

3.2.9 Relative gate speeds

In an effort to quantify the operating speed of a circuit it is necessary to determine the relative delays of the above-mentioned gates. These gates have been simulated using Hspice for similarly sized transistors and output loading in a $0.7\mu\text{m}$ CMOS process, and the *approximate* relative delays are given in Table 3.1 (each delay unit roughly corresponds to 0.15ns in this process).

Gate	Delay	Gate	Delay	Gate	Delay	Gate	Delay
multiplexer	2	inverter	2	nand	3	nor	5
p-nmos xor	6	p-nmos xnor	6	small xor	7	small xnor	9
Svensson latch	6	small latch	7	cgate	10	feed	13
restore	17						

Table 3.1: Relative gate delays in ECS assuming similarly sized transistors and loading.

3.2.10 Micropipeline module exceptions

The *select* and *call* modules in the micropipeline library are not a part of the ECS library of gates. Their functionality can however be reproduced by constructing circuits out of

the more primitive *feed* and *restore* gates.

The most interesting omission from the micropipeline library is that of the *toggle* module, which is there used to alternately steer an input event between two output paths. Its use in SI design paradigms often occurs in situations where a 4P signal (which may have earlier been produced from a 2P conversion) is split into two, 2P signals [Ebe89, FES94, DW95]. In the ECS design framework all control lines utilize 2P signalling only, and therefore the function of the *toggle* module is not required.

Another module which has not yet been considered is the *arbiter*. This important unit ensures that only one of two output event lines is active at any time even if events occur simultaneously on its two input lines. To this end the *arbiter* must resolve its potential internal metastability whilst ensuring that its output states remain valid. Numerous issues dealing with metastability and the implementation of 4P *arbiters* (of which 2P *arbiters* can be built) can be found in [CM73, Den85, EBG93, VBH⁺95].

Instances of arbitration should be avoided, or minimized in frequency, whenever possible, although in some cases the problem is either unavoidable or too costly to avoid (such as in accessing memory from split caches, or incorporating an interrupt request into a control stream). Although the arbitration function is still needed in the ECS paradigm, it is only used in *exceptional* cases, and such occurrences are therefore dealt with outside of the general design framework.

3.3 Some formalisms for gate representations

A library of gates has been presented from which asynchronous 2P circuits can be constructed. These can obviously be represented graphically using the circuit symbols provided, however it is often convenient to represent their functionality and interconnectivity in a more concise and algebraic (textual) form, such as in terms of transitions, logic levels, or event traces.

To this end it is worthwhile investigating some currently popular design methodologies to see how they represent the 2P functionality of the above gates. One can then select a representation which is the most appropriate or, if needs be, devise a new representation.

An important issue to bear in mind is the fact that these methodologies are being analyzed for their *gate-representative* qualities only, such as conciseness (the number of

terms required), complexity (the number of operators required), and indicativeness of transitional 2P behaviour (which is a subjective issue). They are *not* being analyzed from a synthesis perspective as an *engineering* paradigm has already been chosen for attaining the primary goal of high-speed circuitry.

Of the vast array of methodologies possible, the following five have been chosen for consideration: binary logic, Pucknell's transition equations, Chu's STGs, Ebergen's trace specifications, and Martin's CSP production rules. These five approaches are in popular use and exhibit significantly different gate representations.

To illustrate these approaches, a *cgate*, a *send*, and a *feed* gate have been chosen for the analysis. Table 3.2 shows these representations for all but the CSP rules (shown in Table 3.3) and the STG approach (which is shown graphically for clarity in Fig.3.12).

Gate	Binary Logic	Pucknell's TEs	Ebergen's Traces
CGATE	$z = ab + z(a + b)$	$\Delta z > \Delta a \Delta b + \Delta a \nabla b + \Delta b \nabla a$ $\nabla z > \nabla a \nabla b + \nabla a \Delta b + \nabla b \Delta a$	pref*[(a? b?);z!]
SEND	$o = ic + o\bar{c}$	$\Delta o > \Delta c \Delta i + \Delta c \nabla i + \nabla c \Delta i$ $\nabla o > \Delta c \Delta i + \nabla c \nabla i$	pref*[*[(c?) ²];(i? c?); o!;pref*[i?;o!];c?]
FEED	$o = o\bar{c} + c(a \oplus i)$ $a = ac + \bar{c}(o \oplus i)$	$\Delta o > \nabla c \Delta i + \nabla c \nabla i$ $\nabla o > \nabla c \Delta i + \nabla c \nabla i$	pref*[c?;pref*[i?;o!]; c?;*[i?]]

Table 3.2: Representations of the *cgate* (assuming non-prefixed input events of *a* and *b* and an output of *z*), *send*, and *feed* gates (assuming input and output events of *i* and *o* respectively, and a control signal of *c*) in various design methodologies.

CGATE	SEND	FEED
$a \wedge b \rightarrow z \uparrow$	$c \wedge i \rightarrow o \uparrow$	$(i \wedge \neg o \vee \neg i \wedge o) \wedge \neg c \rightarrow a \uparrow$
$\neg a \wedge \neg b \rightarrow z \downarrow$	$c \wedge \neg i \rightarrow o \downarrow$	$(i \wedge o \vee \neg i \wedge \neg o) \wedge \neg c \rightarrow a \downarrow$
		$(i \wedge \neg a \vee \neg i \wedge a) \wedge \neg c \rightarrow o \uparrow$
		$(i \wedge a \vee \neg i \wedge \neg a) \wedge \neg c \rightarrow o \downarrow$

Table 3.3: Representations of the *cgate*, *send*, and *feed* gates using Martin's CSP.

3.4 Analysis of methodologies

The gate expressions in binary logic are comparatively concise but it is difficult to grasp the gate's transitional functionality, particularly for the *feed* gate (which also expresses a micropipeline *select* module). Furthermore, feedback equations are also required which complicates the representation.

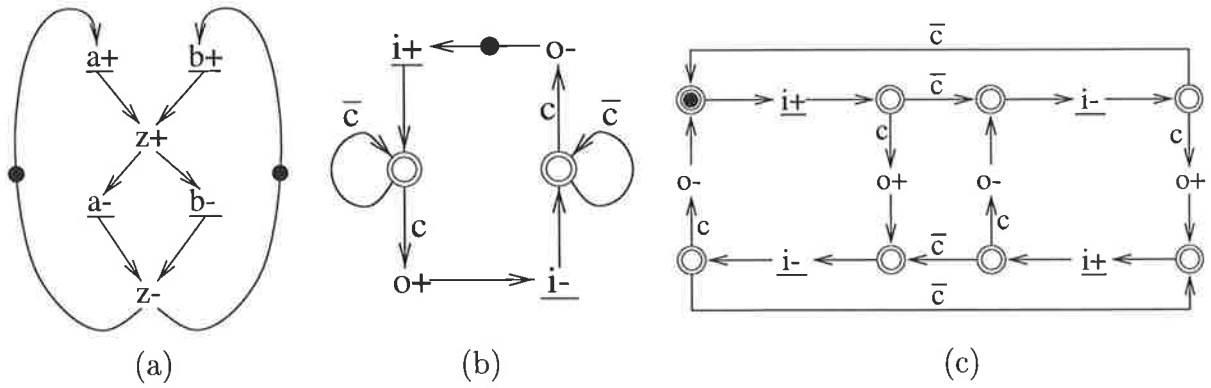


Figure 3.12: Representations of (a) the *cgate*, (b) *send*, and (c) *feed* gates using STGs.

Pucknell’s transition equations express the functionality of a gate in terms of the conditions for positive and negative events. Four states are possible: a transition in either direction or to remain at a logic level. Expressing the gate’s operation in this manner removes the feedback and internal states implicit in the binary representation and also provides an event-based description as desired. However, the increase in the number of states to four results in a less concise format and also means that two equations are now needed for the representation of a gate rather than one (in fact, four equations are required if the “remain at” conditions are also expressed).

Ebergen represents a gate’s allowable traces (the sequence of inputs and outputs) in a concise format using such operations as prefixing (*pref*), repetition (**[]*), sequencing (*;*), and weaving (*||*), with gate signals being defined as either inputs (*?*) or outputs (*!*). This specification is more complex than the previous two descriptions (given the number of operations required), and it is still difficult to grasp the *transitional* behaviour of the gate from the trace specification (particularly for the *feed* and *send* gates, whose conceptual behaviour is particularly straightforward).

CSP can also express the functionality of a gate in terms of its allowable traces (similar to Ebergen’s approach), but is shown at the final gate specification level (involving production rules) in Table 3.3. This representation is highly favourable for the first two gates in which, similar to Pucknell’s transition equations, input conditions are specified for each output event. However in contrast to Pucknell the conditions are given in terms of their binary states which therefore improves its conciseness. The major disadvantage of this representation can be seen in the greater complexity of the *feed* gate, for which the internal states of the gate must again be specified (as in the binary logic representation).

STGs also specify the transitional behaviour of a gate's output and consider the inputs to be transitional as well (unlike CSP which expresses these as binary signals and Pucknell who expresses them in four possible states). An exception to this is when input choice has to be specified, causing logic level signals to be placed on certain arcs and the inclusion of places. Note however that this delineation between data signals and transitioning events is pleasantly indicative of the same distinction which is also made in ECS. Unfortunately however the specification of an ECS gate as an STG is seen to be rather complex and unwieldy, which is especially true when translated into an algebraic (textual) format.

In short then it can be said that none of the above representations are fully suitable in terms of brevity, simplicity, and descriptiveness. Binary logic falls short in descriptiveness, Pucknell's TEs are not concise, Ebergen's traces are neither descriptive nor simplistic, and Martin's CSP and STGs suffer in brevity. Although any of these methodologies could be used (and are used elsewhere) for representing the ECS library of gates, it is worth while investigating new approaches in an attempt to improve upon them. Such an investigation has lead to the ECS representation explained in the following sections.

3.5 The ECS representation

One of the problems encountered in the above methodologies is in modelling an event line's transitory behaviour with logic levels. The positive and negative transitions of this event have been considered independently when in essence the two are identical (as can be seen by the timing diagrams in Section 3.2, in which event line transitions are treated identically regardless of their direction). By somehow combining these two transitions into a single term the complexity of an STG-like representation could be halved, and it may also become possible to implement CSP and TE-like descriptions with a single equation.

Another problem arises from the difference in functionality between data lines and event lines. Since one is state based and the other is transitory the effect of combining the two can introduce excessive complexity (for example, compare the traces of the *feed* and *send* gates against that of the *cgate*). A representation in which the behaviour of these two signal types can be similarly modelled should then improve the conciseness of a gate's

representation and subsequently enable its functionality to be more easily understood.

These are some of the issues dealt with by the ECS representation and which result in the simple, descriptive specification of a gate's operation. In brief, the technique used is to transform the signal types into a new domain (called the *temporal* domain) and introduce some new operators in this domain which then enable concise gate descriptions to be made.

3.5.1 Transforming signals into the temporal domain

Both event lines and data lines have thus far been represented in conventional binary logic, which will henceforth be termed the *voltage* domain (the terms *binary* or *logic* domain are not used because the temporal domain also uses binary logic levels, but which are interpreted differently from those in the voltage domain). In this voltage domain, an event occurs on the signal line ∂in when a transition in either direction occurs:

$$\Delta_v \partial in + \nabla_v \partial in$$

where $\Delta_v \partial in$ evaluates true after a positive edge transition occurs on ∂in in the voltage domain (as indicated by the subscript v), and similarly for $\nabla_v \partial in$, and “+” is the conventional boolean OR operator. As was mentioned before, the actual logic levels ought to be irrelevant, and therefore the direction of the transition ought also be irrelevant, it being merely the occurrence of a transition which is used for control signalling. Therefore, in the ECS representation, a simple transformation is made whereby the occurrence of an event in the voltage domain is converted into a temporal *truth* (ie- a positive transition of the event line) in the temporal domain:

$$\Delta_t \partial in \Leftrightarrow \Delta_v \partial in + \nabla_v \partial in \quad (3.1)$$

where the subscript t indicates the temporal domain and the symbol \Leftrightarrow merely indicates the equivalence of the two statements through the transformation. Note that this transformation *only* applies to event lines. The logic state of a data line (and therefore the direction of any transitions on that line) *is* relevant, and consequently the transformation of these lines into the temporal domain is simply an identity operation (ie- their logical interpretation is unchanged):

$$\Delta_t in \Leftrightarrow \Delta_v in \qquad \nabla_t in \Leftrightarrow \nabla_v in \qquad (3.2)$$

It will be noted then that an event line's transition in the voltage domain is now considered as a binary "1" level in the temporal domain. This then enables a data level to be combined with the *occurrence* of a transition in a simple manner, using conventional logical operators such as *and* and *or*. This reduces the complexity of gate specifications and subsequently improves their descriptiveness.

The temporal transformation for event lines produces an as yet unresolved issue: if an occurrence of a transition on an event line in the voltage domain is used to set its temporal state high, how then is it to be subsequently set low, and furthermore, what are the consequences if, before being set low (by whatever means is devised), another transition occurs in the voltage domain which attempts to force a positive transition when the signal is already (temporally) high??

The first issue to resolve is that of the conditions for producing a $\nabla_t \partial in$ event. One approach is to force this transition immediately after the output of a gate has responded to its input. That is:

$$\nabla_t \partial in \Leftarrow \Delta_t \partial out \qquad (3.3)$$

where ' \Leftarrow ' indicates that a negative transition on ∂in is produced after a positive transition on ∂out (in the temporal domain). ∂out is assumed to be the gate's output event and is generated from ∂in according to the gate equation (discussed in Section 3.5.3). This definition for producing $\nabla_t \partial in$ is referred to as the *Gate Representation* (*G-Rep*), and it can be verified that for all ECS gates the following byproduct results:

$$\nabla_t \partial out \Leftarrow \nabla_t \partial in \qquad (3.4)$$

The event outputs of a gate in the temporal domain will therefore always be pulses (δ functions), the ramifications of which will be discussed in Section 3.5.7. To convert the temporal interpretation of a gate's output back into the voltage domain, the following simple transformation can be used:

$$\Delta_t \partial out \Leftrightarrow \Delta_v \partial out + \nabla_v \partial out \qquad (3.5)$$

which is rather obvious since a logic “1” level in the temporal domain is used to indicate the occurrence of an event in the voltage domain. Fig.3.13 illustrates the different representations of the input and output events of a gate in the voltage and temporal domains, with the solid arrows between signals indicating the equation number responsible for each transition, and the dashed arrows indicating a transition of the (as yet unspecified) gate.

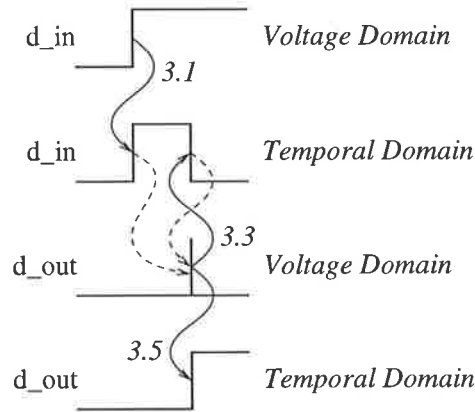


Figure 3.13: An example of the voltage and temporal representations of a gate’s input and output signals, and the equations which cause them.

A framework is now in place whereby the interaction of event line *transitions* and data line *states* are similarly modelled. It is then hoped that the subsequent representation of ECS gates within this framework will become both simple and descriptive, however before this issue can be resolved it is necessary to introduce methods whereby the nature of these interactions can be specified.

3.5.2 ECS operators and temporal equations

Thus far the temporal description of the inputs and outputs of a gate has been presented, however the actual operation of individual ECS gates is still to be addressed. In conventional logic, functions are represented by equating the logical state of an output signal (either high or low) with the logical combination of a group of input signals using the basic functions of *and* and *or*. Similarly, other asynchronous methodologies also employ a collection of predefined operators which are used to specify a gate’s functionality. For example, the *weave* operator (\parallel) in Ebergen’s traces, the *and* operator (\wedge) of CSP, and the positive ($\Delta, \uparrow, +$) and negative ($\nabla, \downarrow, -$) transitional operators used throughout.

In ECS, a similar set of operators are employed to enable the correct specification of ECS gates. Many of these operators are analogous to those used in conventional logic, which is primarily due to the way in which event lines and data lines are modelled. There are however a few additional operators which are introduced to properly delineate between different types of inputs. For example, the restore gate has two input events, however one of these has no effect on the output unless the other has already occurred (ie- they each affect the output in a different manner).

After ‘>’

- Usage: $\mathcal{T}_e > \mathcal{T}_c$

where \mathcal{T}_e is termed the temporal effect and consists of an *assignment* operation (described below), and \mathcal{T}_c is termed the temporal cause.

The after operator is used to govern conditional operations. Specifically, if the temporal cause is true ($\mathcal{T}_c = 1$) then the temporal effect to the left of the after operator is processed. Exactly what this effect is depends on the assignment contained within it.

Assignment (becomes) ‘←’

- Usage: $\mathcal{T}_o \leftarrow \mathcal{T}_i$

Assignment operations are always present to the left of the after operator and specify the function (ie- the temporal effect \mathcal{T}_e as given above) to be performed when the temporal cause evaluates true. It is analogous to the ‘=’ operator in conventional logic, and together with the after operator results in the general specification of a gate (termed a *temporal equation*, or *TE*) as being:

$$\mathcal{T}_o \leftarrow \mathcal{T}_i > \mathcal{T}_c$$

which reads “ \mathcal{T}_o becomes \mathcal{T}_i provided \mathcal{T}_c is true”. This general format enables a broad range of functions to be developed. In particular, conventional binary logic functions may be merged with latching operations in a simple manner. For example:

$$y \leftarrow (a + b) > c.d$$

In this case, the term $c.d$ would be used to govern the latch, with an input of $a + b$ and an output of y . Note also that the analogous TE of a conventional function such as $y = a + b$ may be specified in two identical forms:

$$\begin{array}{ll}
y \leftarrow 1 & > a + b \\
y \leftarrow 0 & > \overline{a + b}
\end{array}
\qquad
y \leftarrow a + b > 1$$

The assignment $y \leftarrow 1$ is abbreviated to just y , and similarly $y \leftarrow 0$ is abbreviated to \bar{y} . The left-hand specification then appears to have the better functional correlation to the conventional form, however it suffers from having to specify two TEs for the gate's functionality, since both the y and \bar{y} terms must be present. The second approach requires only the one TE, however the permanently true temporal cause seems like overkill.

As a better solution, the convention is adopted in which if no after term is specified, then “> 1” is assumed. Therefore specifying $y \leftarrow a + b$ is identical to specifying $y \leftarrow a + b > 1$. Note that one could adopt the convention whereby specifying $y > a + b$ implies the existence of the converse TE: $\bar{y} > \overline{a + b}$, however this approach still essentially deals with two TEs rather than one.

Consider now the general specification of a TE in which all temporal signals are events:

$$\mathcal{T}_o^E \leftarrow \mathcal{T}_i^E > \mathcal{T}_c^E$$

In this instance, \mathcal{T}_c^E is termed a *causal input* event, \mathcal{T}_i^E is termed an *effectual input* event, and \mathcal{T}_o^E is termed the *effectual output* event. By virtue of the fundamental premise of the after operator, if \mathcal{T}_c^E is false (ie- no causal input event has occurred) then an event (a temporal truth) which occurs on \mathcal{T}_i^E will not be processed.

This issue can present difficulties when one considers that the temporal transformation used for event lines requires that each input event must have a corresponding output event. For \mathcal{T}_c^E this is given by \mathcal{T}_o^E , however it is not generally acceptable to define \mathcal{T}_o^E as also being the output event for \mathcal{T}_i^E , since when $\mathcal{T}_c^E = 0$ this term is not processed, and consequently no output event occurs. Therefore only one such event (in the voltage domain) would be permitted to occur on \mathcal{T}_i^E (which then sets its temporal signal high) even when $\mathcal{T}_c^E = 0$ when ideally, since the assignment isn't processed, any number of events ought to be allowed to occur.

One solution to the problem is to define an attempt to force $\Delta\mathcal{T}_i^E$ when \mathcal{T}_i^E is already high as having no effect, however this is unacceptable as it's equivalent to making \mathcal{T}_i^E a causal term and therefore prevents some ECS gates (such as *feed* and *restore*) from

being modelled. Furthermore, this then prevents the incorporation of error checking into the ECS representation (see Section 3.5.8). The correct solution is to define \mathcal{T}_i^E as its own output event, so that its temporal input to a gate always appears as a δ pulse. This definition then enables any number of events to occur on \mathcal{T}_i^E independently of the temporal state of \mathcal{T}_c^E , which therefore enables the feed and restore gates to be properly modelled. Note however that if \mathcal{T}_i^E is in fact a combinational term of input events (such as $\partial a.\partial b$), then it is the combinational term which is the output event for these signals, not the individual signals themselves. In essence, such a combinational term is treated as its own enclosed TE.

Logical AND (‘.’) and OR(‘+’)

- Usage: $\mathcal{T}_{ic1} \cdot \mathcal{T}_{ic2}$ or $\mathcal{T}_{ic1} + \mathcal{T}_{ic2}$

The “and” and “or” logic functions are analogous to those used in conventional logic.

Until ‘U’

- Usage: $\mathcal{T}_{ic1}^E U \mathcal{T}_{ic2}^E$

The until operator has no equivalent function in conventional logic because it is used explicitly on event lines. The result of the operation is temporally true when \mathcal{T}_{ic1}^E evaluates true and computes false when \mathcal{T}_{ic2}^E evaluates true. On this basis, the result of an until operation is effectively a data signal which is used to govern the effect of a TE. For example, the following two temporal specifications (which are a collection of one or more TEs, and dubbed a TS), are equivalent:

$$y \leftarrow x > \partial pass U \partial hold \qquad \qquad \qquad select \leftarrow \partial pass U \partial hold$$

$$y \leftarrow x > select$$

Note also that to facilitate ease of implementation, the event \mathcal{T}_{ic2}^E is defined to be the output event of \mathcal{T}_{ic1}^E , and vice versa. Defining each signal to be its own output event would be equally valid (and in fact produces a more versatile gate representation) but doing so results in a more complex implementation, which is unjustified since almost every application of this gate uses alternating events (hence the restriction). Note that a non-alternating event stream could simply be masked with a *feed* gate controlled by the result of the until operation to transform it into an alternating one.

Colon ‘:’

- Usage: $\mathcal{T}_1 : \mathcal{T}_2 : \dots : \mathcal{T}_n$

The colon operator performs no logical operations, but rather signifies the validity of an output signal \mathcal{T}_1 which results from some earlier input signal \mathcal{T}_n . That is, it specifies a computation path which exists between these signals, and ought therefore be considered as a conceptual timing operator. It is often used to specify the requirements of a lumped delay element, as exemplified by the TE below:

$$\partial z : \partial y : \partial x \leftarrow a : b : c : \partial x$$

In this instance, the event ∂x results in the data signal a being set as well as independently resulting (through some other path) in an event on ∂z (the TEs for these operations would be given elsewhere in the TS). The colon operator indicates that a should be valid before ∂z occurs, and consequently a delay may have to be inserted from ∂x to ∂z . Typically, another TE would exist in the TS in which an output depends on both ∂z and the new value of a (otherwise no such timing constraint would need to be specified). Note also that specifying the intermediate signals ∂y , b , and c is optional, and is included only to highlight the relevant control and computational paths.

3.5.3 Some ECS gate examples

The set of operators just described can be used to combine both data and event lines into a collection of TEs which each correspond to one (or more) of the gates in the ECS library. Note however that TEs which are constructed with an output event that can occur without an input event (and vice versa) are prohibited, as these violate the requirements imposed by the temporal transformation. For example, a TE which *or*'s an event line and a data line is prohibited, as are TEs which assign data lines to event lines (or vice versa).

The complete set of fundamental TEs and their corresponding ECS gates is given in Appendix A. Example waveforms for the *send* and *feed* gates are shown in Figs.3.14a and 3.14b respectively to illustrate the different representations of signals in the voltage and temporal domains.

Consider the first *din* event which occurs. For the *send* gate, the occurrence of this event is transformed into a logical one in the temporal domain, which when *and*'ed with

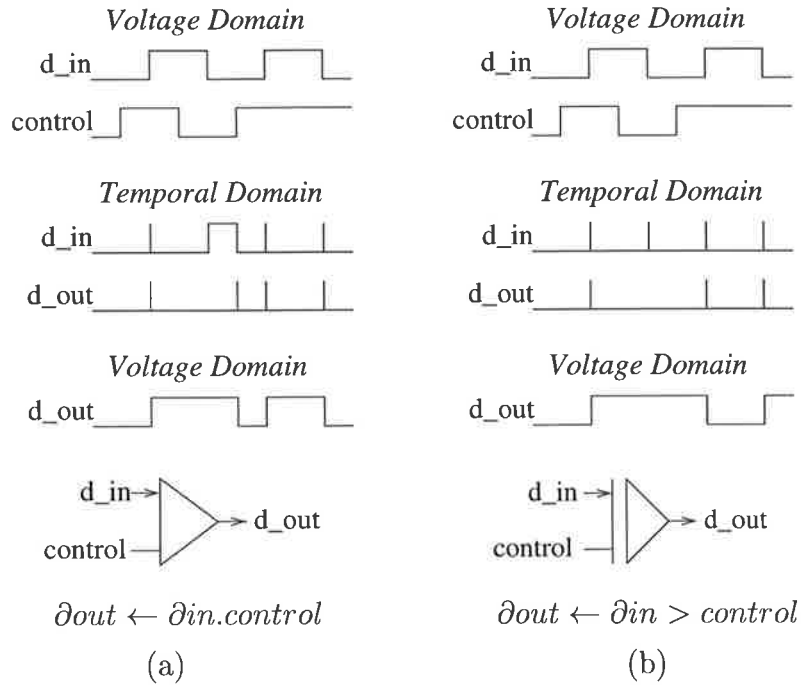


Figure 3.14: Waveforms for the (a) *send* and (b) *feed* gates in the voltage and temporal domains.

$control = 1$ (as required by the TE) results in the effectual input term of the TE being high. This is assigned to the temporal state of the output ∂out (since the causal term is permanently true), and by virtue of Eq.3.3 this then forces the temporal state of ∂in low again. The effectual input term is now false, and therefore $\nabla \partial out$ then occurs.

For the *feed* gate, the occurrence of an event in the voltage domain sets ∂in temporally high, and since the causal term $control$ is also high, this gets assigned to ∂out . Since ∂in is an effectual input event (and therefore acts as its own output event), $\nabla \partial in$ occurs immediately, which subsequently sets the state of ∂out low again. Therefore both gates generate a ∂out transition for the first event on ∂in .

With the second event however things are somewhat different. For the *send* gate, the combined term $\partial in \cdot control$ remains low until the $control$ signal goes high. At this point the output also goes high and as before then results in $\nabla \partial in$ and $\nabla \partial out$. For the *feed* gate however the causal term is low. As such, the assignment on the left is not processed when $\Delta \partial in$ occurs, and since it acts as its own output event, $\nabla \partial in$ occurs immediately thereafter. It is evident then that in this instance, no output is generated on ∂out . The analysis for the third and fourth input events are identical to the first. Referring to the waveforms in the temporal domain it can be seen that the output events from both gates are in the form of infinitesimal pulses (assuming zero gate delay).

The results of the analysis performed in the temporal domain may be converted into the voltage domain by the simple inverse transformation given in Eq.3.5. It can be seen that the resulting output waveforms for the *send* and *feed* gates in the voltage domain are identical to those shown in Figs.3.4b and 3.6c respectively, which indicates that the TEs given above do in fact represent the required logical functionality of these gates.

To further exemplify the specification and functionality of the ECS gates in the temporal domain, the TEs and timing diagrams for the *cgate* and a *latch* are given in Figs.3.15a and 3.15b respectively. Note that for the *cgate*, ∂out is the corresponding effectual output event of both ∂ina and ∂inb . It is a simple matter to follow the analysis above and thereby demonstrate that the TEs again properly represent the gate's logical functionality.

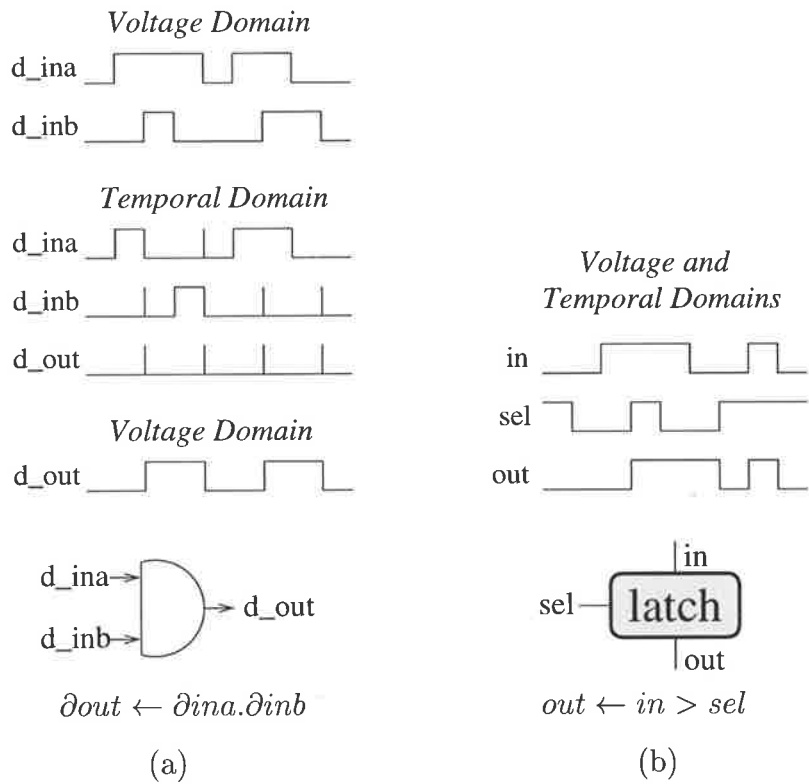


Figure 3.15: Waveforms for (a) the *cgate* and (b) a *latch* in the voltage and temporal domains.

In summary, the representation of ECS gates consists of the following steps:

1. Transform the input signals into the temporal domain using Eqs.3.1 and 3.2.
2. Determine the causality of the TE and hence assign the state of the output.
3. If high, reset the state of the event inputs using Eq.3.3, and re-assess the output.
4. Convert the temporal output into the voltage domain using Eq.3.5.

3.5.4 Comparative gate representations

By comparing the gate specifications given in Section 3.3 for the *send*, *feed*, and *agate* against those given in the previous section using the ECS representation, it becomes evident that a significant improvement has occurred. For every ECS gate only one equation is now required to represent its functionality, whereas at least two are required in general in all other paradigms except Ebergen's.

Another advantage of the ECS representation is in the conciseness of the description. Only a few terms are needed in general to represent a gate in ECS whereas many more are needed in all of the methodologies considered earlier. This has the subsequent advantage of also making the representation more readable and understandable. This is especially evident in the TEs for the *agate* and *merge* gate, which are often dubbed the *and* and *or* functions to events in the literature. None of the other paradigms mirror this effect, whereas in the ECS representation these are the *exact* same operations employed in their specification. This ECS representation is therefore adopted for the future specification of 2P asynchronous circuits.

3.5.5 Some example TS's and their corresponding circuits

Fig.3.16 shows an event control circuit for one stage of a reconfigurable FIFO [MAL94]. This circuit operates such that when the control signal *stage* is high, an incoming event from the preceding stage is fed to the output *dfifodone*. This prevents any further stages from triggering, and hence the FIFO length is given by the number of stages preceding it. A return event (from whatever circuit the FIFO output triggers) is fed back via the *feed* and *merge* gates to complete the handshaking with the previous stage. If however the control signal *stage* is low, then the event control simply passes through to the next stage.

The important point to note with regard to the methodology is the correlation between each TE and its ECS gate implementation in the circuit. For example, the last TE corresponds to the *merge* gate immediately prior to the handshaking with the previous stage. It is therefore evident that the TS is representative of the circuit topology, and specifies this in a clear and concise format.

Another example of the correspondence between a TS and its circuit topology is shown

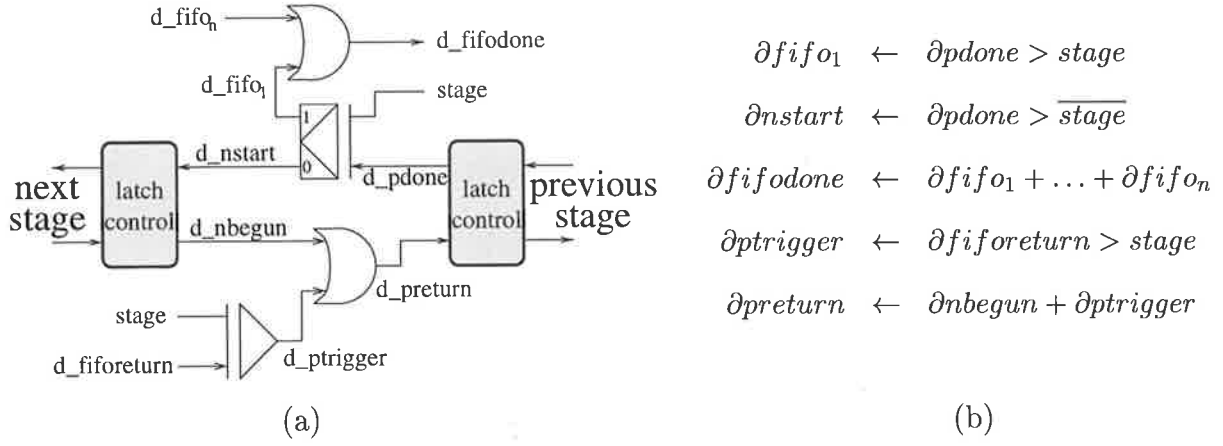


Figure 3.16: (a) Circuit design and (b) the corresponding TS of a reconfigurable FIFO.

in Fig.3.17, which shows a *slice* of the input control circuitry for a Discrete Fourier Transform (DFT) unit [MAL94]. Again, the simplicity and readability of the TS, and its equivalence to the control circuit, is evident. Note also that the overline on $\overline{\partial f i f o r e t u r n}$ is used in the TS to indicate a primed event and *not* an inverse of the event in the temporal domain (which is non-sensical).

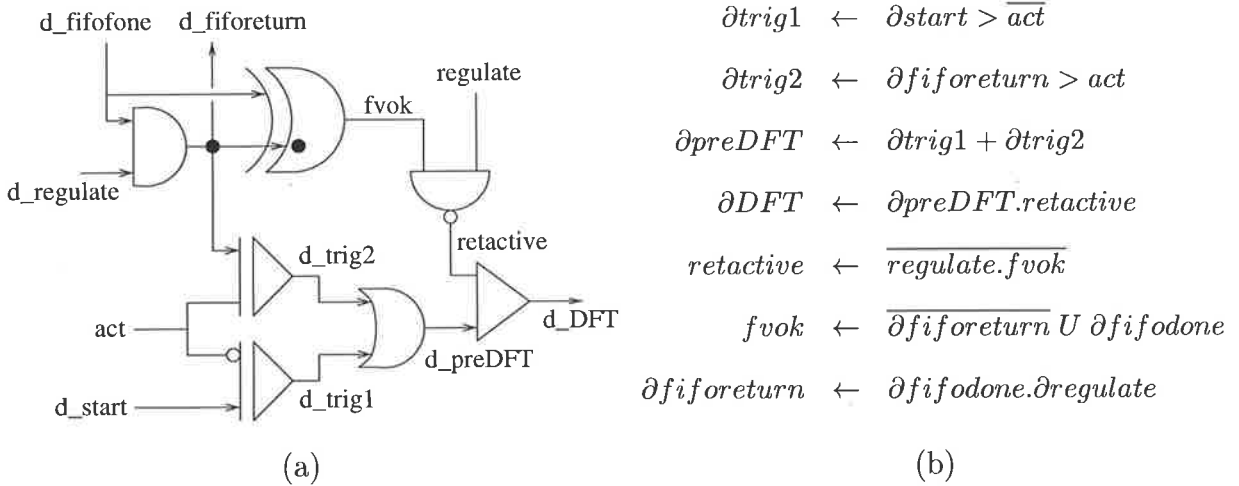


Figure 3.17: (a) Circuit design and (b) the corresponding TS of a slice of the input circuitry for a DFT unit.

It should also be noted that TEs can be combined (or split) to produce a temporal specification with fewer, but more complex, terms (or the converse when splitting TEs). For example, the 3rd through 6th TEs of Fig.3.17b could be combined and re-written as:

$$\begin{aligned}
 \partial DFT &\leftarrow (\partial trig1 + \partial trig2).(\overline{regulate}.(\overline{\partial f i f o r e t u r n} U \partial f i f o d o n e)) \\
 &\leftarrow (\partial trig1 + \partial trig2).(\overline{regulate} + (\partial f i f o d o n e U \overline{\partial f i f o r e t u r n}))
 \end{aligned}$$

Combining and splitting TEs can help produce a TS which is faster and/or smaller than initially conceived (see Section 4.1).

3.5.6 Precedence and properties of temporal operators

The order in which the temporal operators are processed is defined in Table 3.4, in which the operator precedence decreases from left to right.

Precedence	Highest → Lowest					
Operator	And	Or	Colon	Until	Assignment	After
Symbol	.	+	:	<i>U</i>	←	>

Table 3.4: Order of precedence for the ECS operators.

For example, the TE shown to the left below could equally well be written as per the TE on the right with the brackets removed, given the precedence rules above.

$$(sel \leftarrow ((a.b) + c)) > ((\partial x.d) U (\partial y : \partial z)) \quad sel \leftarrow a.b + c > \partial x.d U \partial y : \partial z$$

The associative, commutative, and distributive properties of the and, or, and until operators are summarized in Table 3.5. For the “and” and “or” operations these properties are almost identical to their binary logic counterparts, however deviations exist when event lines are considered. Note that the colon, assignment, and after operators do not exhibit such properties.

Signal Types	data / data			data / event			event / event		
Property	Or	And	Until	Or	And	Until	Or	And	Until
Associativity	✓	✓	-	-	✓	-	✓	✓	-
Commutivity	✓	✓	-	-	✓	-	✓	✓	x
Distributivity	✓	✓	-	x	(d,e,e)	-	over AND	x	x

Table 3.5: Various properties of the ECS operators.

A tick (✓) or a cross (x) in the relevant box means that the property is valid or invalid respectively, and a dash (-) means that the property has no meaning because the TE is erroneous. The special distributive conditions “(d,e,e)” and “over AND” respectively imply the following rules:

$$\begin{aligned}
a.(\partial b + \partial c) &= a.\partial b + a.\partial c & \partial a + \partial b.\partial c &= (\partial a + \partial b).(\partial a + \partial c) \\
a.(\partial b.\partial c) &= (a.\partial b).(a.\partial c)
\end{aligned}$$

By manipulating TEs according to the above properties it becomes possible to reduce circuit complexity and increase system speed algebraically (see Section 4.1).

3.5.7 Interconnectivity of gates in ECS

By considering the timing diagrams of the gates in the temporal domain, it appears that two problems may exist with regard to connecting them together to form circuits. Firstly, if the temporal outputs of a gate are always pulses, then how is it possible for a subsequent gate (such as a *send* or a *cgate*) to function properly when the input is required to be kept pending (and therefore kept high in the temporal domain)? Secondly, if the output of a gate is determined by its inputs, and yet when connected to a subsequent gate as an input its temporal level can be set low via Eq.3.3, then is there not a conflict with regard to driving the signal level on this line?

To overcome these problems an explicit ECS definition is made for the *global* view of the event line output of a gate (data line outputs do not suffer from these problems). The configuration of gates shown in Fig.3.18 illustrates this principle.

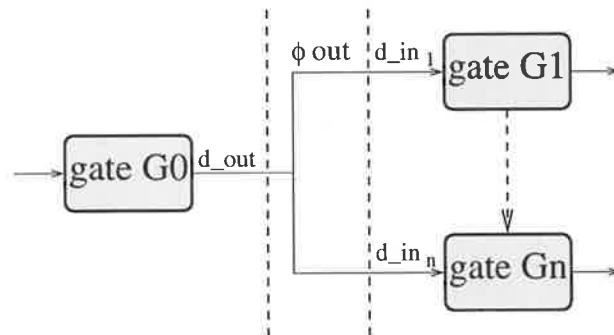


Figure 3.18: A ϕ event line which connects a gate's output to the inputs of subsequent gates.

The local output of *source* gate G_0 (∂out) is connected to the global event line ϕout . This is then distributed to each of the subsequent *sink* gates $G_1 \dots G_n$, where it then connects onto each gate's local input ($\partial in_{1\dots n}$). The temporal state of ϕout is given by the temporal signal ∂out as well as the input signals $\partial in_{1\dots n}$ of each sink gate according to the following equations:

$$\Delta\phi_{out} > \Delta\partial_{out} \quad (3.6)$$

$$\nabla\phi_{out} > \nabla\partial_{in_1} \cdot \nabla\partial_{in_2} \dots \nabla\partial_{in_n} \quad (3.7)$$

with the temporal state of a sinks input being given by $\Delta\partial_{in} > \Delta\phi_{out}$ (remembering that $\nabla\partial_{in}$ is then set locally according to the gate's TE, and in particular, the firing of its corresponding output event). Equation 3.6 propagates the truth of an event *only* from the source, as this indicates that an event has occurred in the voltage domain, and equation 3.7 indicates when all subsequent gates have responded to this input and therefore a new output is permitted to occur.

An alternative to the introduction of global event lines (dubbed ϕ event lines) is to convert back to the voltage domain at each gate's output, then propagate this to each of the subsequent gate's inputs and then transform back into the temporal domain. This approach however is somewhat unsavoury in that it requires a mixed voltage/temporal interpretation.

3.5.8 Principles of error detection

It is now apparent that a means for error detection is possible. If an attempt is made to force a ϕ event line high when it is already high then an error has occurred (similarly for an attempt to force low an already low event line). This error is dubbed an *exclusion violation* (EV) and represents the case whereby two successive events in the voltage domain have occurred at the input of a gate before a corresponding output event has occurred.

With the infinitesimal gate delays shown thus far, an EV error can only occur for the *send* and *cgate*, however if gate delays are now introduced into the model, then any gate can exhibit this error when the time between inputs is less than the gate's propagation delay (t_{prop}). For example, in the timing diagrams of the *send* and *merge* gates of Fig.3.19, an EV error is detected on the third input event. This assumes an inertial delay model for both the positive and negative transitions of the input events in the temporal domain, which means that the positive transition of ∂_{ina} for the *merge* gate is still queued to effect $\Delta\partial_{out}$, even though it gets reset low before this can occur. Note that the $\nabla\partial_{ina}$ transition results from Eq.3.3 and is not affected by the gate delay model.

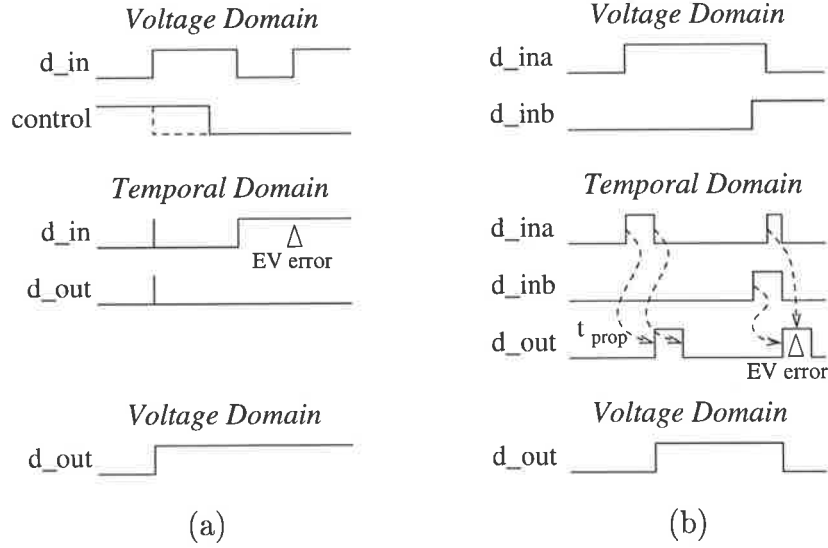


Figure 3.19: (a) A *send* gate with infinitesimal delay, and (b) a *merge* gate with a finite delay t_{prop} , both exhibiting EV errors.

Incorporating propagation delays into the temporal representation merely involves delaying the assignment operation in the TE by the given amount, as is indicated by the first output event of Fig.3.19b. This then enables system speeds to be modelled in parallel with the detection of system errors.

Consider also the *send* gate of Fig.3.19a in the situation where the control signal goes low *at the same time* as the first input event occurs (as shown by the dashed line). In theory the *and* term of the TE will remain low and hence no output event will occur, however in practice signals exhibit finite rise and fall times, so that as long as these two signals change state within a certain time frame, the output will in fact be indeterminate (ie- metastability exists). This situation is known as a *coincidence violation* (CV).

To enable this violation to be detected each gate must also be given a slope time t_s for its output (although a global slope time applicable to all gates may also be assumed for simplicity), which is used to represent the gate's finite rise and fall times.

Given that two input signals \mathcal{I}_1 and \mathcal{I}_2 transition (temporally) at times t_1 and t_2 (with $t_1 \leq t_2$) and have slope times of t_{s1} and t_{s2} respectively (Fig.3.20b), then if $t_1 + t_{s1}/2 > t_2 - t_{s2}/2$ (which indicates that their slopes overlap) a potential CV error exists, and can be resolved as follows:

- Let $t_{a1} = t_1 - t_{s1}/2$, $t_{b1} = t_2 - t_{s2}/2$, $t_{a2} = t_2 + t_{s2}/2$, and $t_{b2} = t_1 + t_{s1}/2$, and define condition \mathcal{C}_a such that the transitions on \mathcal{I}_1 and \mathcal{I}_2 occur instead at times t_{a1} and t_{a2} , and similarly for condition \mathcal{C}_b but with times t_{b1} and t_{b2} .

A CV error is then detected when the output of the gate (given by its TE) for C_a is different to the output which results for C_b . Note that if \mathcal{I}_1 or \mathcal{I}_2 are event lines then a CV only needs to be investigated for a positive temporal transition of the signal.

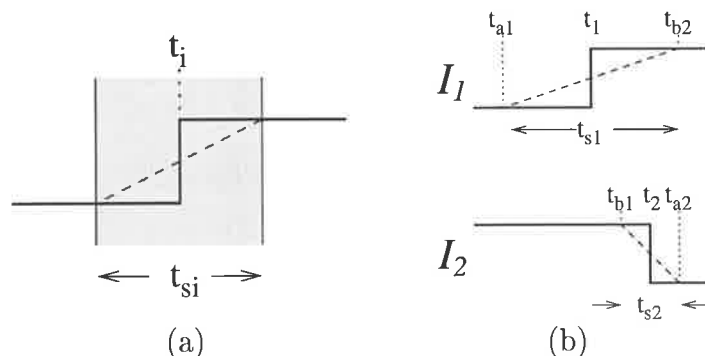


Figure 3.20: (a) A transition t_i with a *shaded* slope time t_{si} , and (b) two transitions which may exhibit metastability.

This technique essentially considers an instantaneous transition at time t_i as occurring (still instantaneously) at either extreme of its slope time t_{si} , as shown in Fig.3.20a. If a transition on another input line (also considered at its extremes) occurs within this region, then the gate needs to be checked for indeterminacy. By assuming that the transitions *could* occur at their extremes, and then checking the resultant output for both cases, the potential for metastability (when these are different) is revealed.

When applied to data nodes a CV returns the occurrence of a potential glitch whose duration may be as long as the overlapping region. Fig.3.21 illustrates this situation for a 2-input *and* gate.

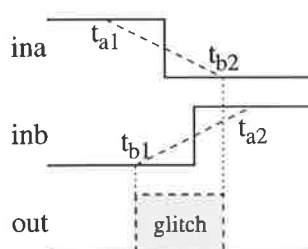


Figure 3.21: Example of a potential glitch for the *and* gate.

Although the output of this gate should remain low, a consideration of the slope times of the inputs shows otherwise. For condition C_a (with transitions occurring at times t_{a1} and t_{a2}) the output of the gate does in fact stay low, however for condition C_b (with times t_{b1} and t_{b2}) the output goes high for a duration equal to the overlap time, indicating that

a glitch may occur. Glitching of data nodes may not however be of detriment to the system (especially when in the data path) and should therefore be treated as *warnings* rather than *errors*.

3.6 Summary

This chapter has introduced a library of asynchronous gates which are used in the construction of ECS circuits. Interestingly, these gates have been shown to be a primitive set of the asynchronous modules used in other design methodologies.

The ECS representation which has spawned this gate library enables them to be expressed algebraically in a clear and concise form using the temporal transformation. It should be noted however that this transformation merely enables a concise and descriptive representation to be developed (as well as incorporating principles of error checking) and is not used to synthesize ECS circuits. This latter issue is discussed in the following three chapters which provide a range of novel engineering techniques for the design of high speed asynchronous systems.

Chapter 4

Fast Asynchronous Circuit Techniques

THE ECS design methodology has thus far provided for a unique set of fundamental gates, a simple and convenient approach to their representation, and a means by which circuit errors can be identified. With this basis it is possible to construct simple two phase, bounded delay circuits, however without a knowledge of advanced techniques these are likely to be inefficient with regards to speed, and most likely with regards to other factors such as power dissipation and area too.

It is therefore necessary to explain in detail the range of design techniques developed by the author which enable high speed two phase asynchronous circuits to be implemented. Although high speed operation is the focus of the techniques described in this chapter, an emphasis has also been placed on the area and power dissipation of the resulting ECS circuits, and in particular how they compare against corresponding SI design approaches.

4.1 Algebraic improvements of a TS

The most formal method for improving the speed of a design is through algebraic manipulation. This involves substituting the *terms* of a TE (to the right of the assignment operator) with the TE definitions for those signals appearing elsewhere in the TS. This newly formed TE is termed *complex*, because it now provides a description for the signal which requires more than one gate in its implementation. By then massaging this complex TE with the laws of associativity, commutivity, and distributivity given in Section 3.5.6,

it is often possible to devise a new implementation which, for the majority of conditions, is faster. Determining whether or not this is the case usually requires an understanding of relative gate speeds (as estimated in Table 3.1) as well as the environmental input constraints.

As a simple example of this process, consider one implementation of a gated pulse circuit and its TS as shown in Fig.4.1. This circuit is used to prevent the activation of a self-timed pulse until some governing condition *go* becomes true.

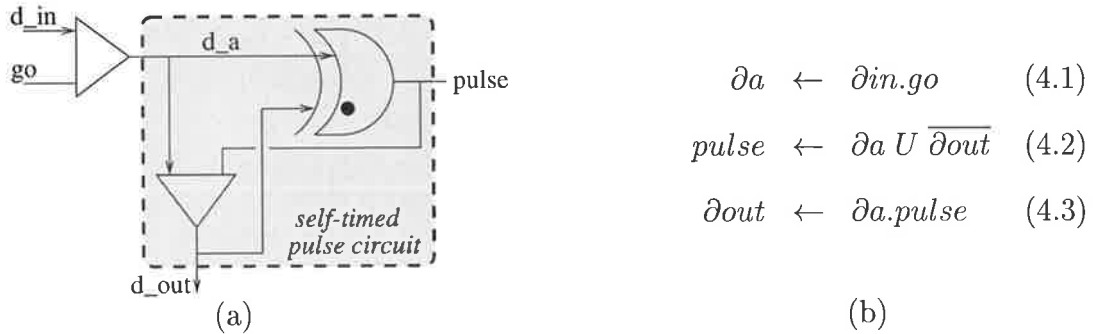


Figure 4.1: (a) Circuit design and (b) the corresponding TS of a gated pulse circuit.

Temporal equations 4.1 and 4.3 can be combined to produce a complex TE for ∂out :

$$\partial out \leftarrow (\partial in.go).pulse$$

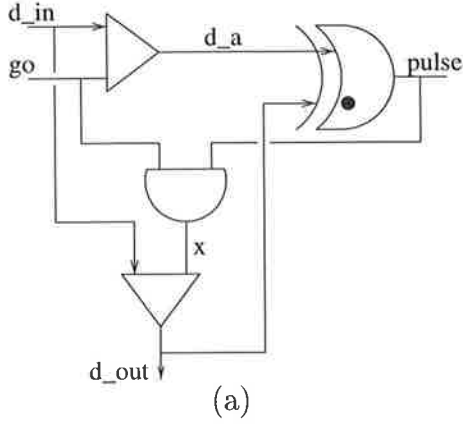
The brackets merely indicate the substitution made, and due to the associativity of the *and* operator for both events and data, can equivalently be placed around the logic signals, giving:

$$\partial out \leftarrow \partial in.(go.pulse)$$

Renaming this *and* term as a TE for *x* then results in the circuit implementation and TS of Fig.4.2. Note that the TE for ∂a is still included in the TS, as it's also used in the generation of the *pulse* signal.

At this level of substitution it can be seen that the circuit speed has in fact been *slowed* by the delay of the *and* gate, and as such is a worse implementation than that of Fig.4.1. However, the substitution process need not stop here. TE 4.4 can be substituted into TE 4.5 to produce a complex TE for the *pulse* signal:

$$pulse \leftarrow (\partial in.go) U \overline{\partial out}$$



$$\begin{aligned} \partial a &\leftarrow \partial in.go & (4.4) \\ pulse &\leftarrow \partial a U \overline{\partial out} & (4.5) \\ x &\leftarrow go.pulse & (4.6) \\ \partial out &\leftarrow \partial in.x & (4.7) \end{aligned}$$

(b)

Figure 4.2: (a) Circuit design and (b) the corresponding TS of a degraded gated pulse circuit.

Assuming now that the signal go does not transition low again until the event ∂out has occurred (which places a constraint upon the operation of the environment, albeit one which is invariably obeyed), then the TE for $pulse$ can be re-written as:

$$pulse \leftarrow (\partial in U \overline{\partial out}).go$$

A simple evaluation of the traces will reveal this equivalence. An important point to note now is that the event ∂a is no longer used in the TS, and its TE can therefore be removed (thus reducing complexity and increasing system speed).

Further, the *until* term of the TE for $pulse$ can be renamed as a TE for p ($p \leftarrow \partial in U \overline{\partial out}$), and by substituting the new TE above for $pulse$ into TE 4.6, the signal x can be reduced:

$$\begin{aligned} x &\leftarrow go.pulse \\ &\leftarrow go.(p.go) \\ &\leftarrow go.p \\ &\leftarrow pulse \end{aligned}$$

Consequently the signal x is no longer required in the TS, and can instead be substituted by the $pulse$ signal. The final circuit and TS which then results is given in Fig.4.3.

Table 4.1 summarizes the system speeds (from ∂in to ∂out) for the above three implementations of the gated pulse circuit, for the two cases in which ∂in occurs both before and after Δgo (the relative gate speeds are taken from Table 3.1, and assume $t_{and} = t_{nand} + t_{inv}$).

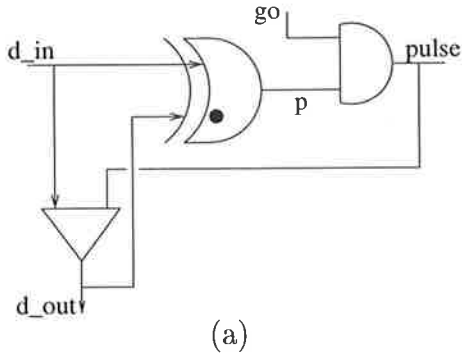


Figure 4.3: (a) Circuit design and (b) the corresponding TS of an improved gated pulse circuit.

Circuit Implementation	$\Delta go < \partial in$		$\partial in < \Delta go$	
	Gate Delay Path	Delay	Gate Delay Path	Delay
Initial	$2t_{send} + t_{xor}$	19	$2t_{send} + t_{xor}$	19
Degraded	$2t_{send} + t_{xor} + t_{and}$	24	$2t_{send} + t_{xor} + t_{and}$	24
Improved	$t_{send} + t_{xor} + t_{and}$	18	$t_{send} + t_{and}$	11

Table 4.1: Relative speeds of three gated pulse circuits.

It is evident that the circuit of Fig.4.3 provides the fastest system speed, with an improvement of up to 42% over the initial implementation (note also that an improved *pulse* signal can be generated with this circuit due to the *nand* and *inverter* driver). As such it is evidenced that through the use of algebraic manipulation a faster TS can be achieved.

4.1.1 Useless TE substitutions

In the preceding example there was no mention of constraints on whether or not a TE can be substituted into another, however there is in fact one important rule which should be obeyed: any TE with an explicit *after* ($>$) clause should not be substituted into another TE. This is because the assignment to the left of the operator is only valid under certain conditions (as given to the right of the operator) and these cannot be separated without altering the initial functionality. As such, there can be no algebraic manipulation of this term and hence, the substitution is futile. For example, substituting TE 4.11 below into TE 4.12 to produce TE 4.13 is redundant, since no simplifications can be made to this term.

$$\partial a \leftarrow \partial b > c \quad (4.11)$$

$$\partial c \leftarrow \partial a.\partial e \quad (4.12)$$

$$\partial c \leftarrow \partial e.(\partial b > c) \quad (4.13)$$

4.1.2 Useful TE substitutions

One of the most commonly occurring improvements which can be made is driving an event through two send gates driven by different control signals. By combining the two TEs as shown from the left below, and then splitting them into a *send* gate and an *and* gate as shown by the TS on the right, a faster implementation will result. At best, an improvement of t_{send} can be gained (a 50 % speed increase), and at worst an improvement of $t_{send} - t_{and}$ (a marginal increase). Note however that this reduction technique requires that ∇x doesn't occur before y goes high (a constraint on the environment), otherwise the *and* of these signals will not produce a ∂c event which would have been produced in the initial TS.

$$\begin{array}{ccc} \partial b \leftarrow \partial a.x & & z \leftarrow x.y \\ \partial c \leftarrow \partial b.y & \xrightarrow{\text{combine}} & \partial c \leftarrow \partial a.x.y \xrightarrow{\text{split}} & \partial c \leftarrow \partial a.z \end{array}$$

Another useful improvement involves simplifying a *merge* gate driven by two common *send* gates (as described by the left-hand TS below). By substituting the top two TE's into the TE for ∂FU , and then factoring out the common *valid* signal, the smaller and faster TS given on the right results. In this instance, a best case improvement of t_{merge} (over 50% speed increase) is the result.

$$\begin{array}{ccc} \partial av \leftarrow \partial a.valid & & \partial ab \leftarrow \partial a + \partial b \\ \partial bv \leftarrow \partial b.valid & \xrightarrow{\text{combine and split}} & \partial FU \leftarrow \partial ab.valid \\ \partial FU \leftarrow \partial av + \partial bv & & \end{array}$$

4.1.3 Taking advantage of the *typical* scenario

In some instances a complex TE can be split in a variety of ways which at first glance may all seem equivalent in terms of speed. However it is often possible to choose one such implementation which provides the greatest speed for what is anticipated to be the most common case (either by proof or intuition). As an example, consider the TE $\partial ok \leftarrow \partial a.\partial b.begin$, which can be split into either of the two TS's shown below:

$$\begin{array}{cc} \partial int \leftarrow \partial a.\partial b & \partial int \leftarrow \partial a.begin \\ \partial ok \leftarrow \partial int.begin & \partial ok \leftarrow \partial b.\partial int \end{array}$$

Although both circuits employ the same number of gates and the same worst case delay ($t_{send} + t_{cgate}$), their best case speeds can be substantially different depending on the usual signalling sequence. If Δ_{begin} is most likely to occur after both of the input events, then the left-hand TS will give the better typical performance (of just t_{send}), however if it's likely to occur before the input events, or if one event (say, ∂b) is usually the last to occur, then the right-hand TS will give the better circuit speed (of t_{cgate}).

It is clear then that although some algebraic reductions will for all cases give improved speed performance (as per the gated pulse circuit example given previously), others will however require a greater understanding of the typical environmental conditions to produce the fastest circuit implementation.

Another example of this is a multiple input *merge* gate, given by the TE: $\partial out \leftarrow \partial a + \partial b + \partial c + \partial e$. Although for a random trace the best approach would be to use a binary tree of 2-input *merge* gates (giving an average delay of 2 gates), if it is known that one of these events (say, ∂a) occurs most frequently, then a *chain* of gates may give the best typical performance (with ∂a triggering the last of these). Similar arguments also apply to a multiple input *cgate*, when it is known that one of the inputs is typically the *last* to occur.

4.2 Improving acknowledge times

One of the most fundamental causes of delay in a SI system is the overhead introduced through acknowledgements. For a SI system to be safely composed from modular blocks, each block *must* ensure that it has reached a stable, final state before sending an acknowledgement back to its environment [BE89] (this, as much as anything, results from the definition of speed independence). The latency (from input to output) imposed by any one block can be quite substantial, and this in turn can increase the overall cycle time of the system since any subsequent, dependent blocks cannot be activated until they receive this acknowledgement.

Consider for example a typical event controlled SI system as shown in Fig.4.4, where λ indicates the event latency for each unit (from input event to output event), and d indicates the data latency (from input event to worst case output data):

The cycle time for this system (taken as the sum of the latencies) is $\sigma = 36ns$,

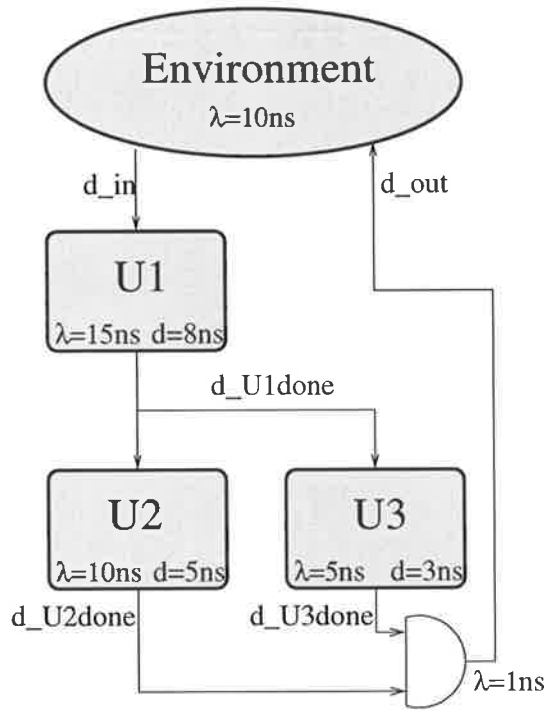


Figure 4.4: An example system constructed in a SI environment.

and cannot be reduced if one assumes a speed independent paradigm. This is because the latency information for each block cannot be utilized, and instead the provision for unbounded gate delays must be adhered to. Therefore theoretically, the figures for λ and d must still be considered to be anywhere in the range from $0 \rightarrow \infty$!

In practice however such extreme variations in gate delays almost never occur, and they can in fact be reliably quantified (as evidenced by synchronous designs). Therefore in an ECS environment, common-sense improvements can be made to the above system to improve the cycle time (albeit at a very high level of abstraction, involving just the interconnection strategy).

One technique is to *remove unnecessary acknowledgements*. It can be seen from the latencies that $U2$ will always produce its output event after $U3$, and therefore $\partial U2done$ can be used to feed directly into ∂out (discarding $\partial U3done$) without having to go through the *cgate*. This approach cannot be used in a SI environment because, however unlikely, the possibility of $\partial U3done$ occurring after $\partial U2done$ must still be provided for.

Another technique involves *providing an earlier acknowledgement* back to the environment, and requires that the settling time of any subsequently triggered unit is less than the resulting cycle time. Consider for example $U2$ in the above system. Once triggered $U2$ takes at most 10ns to settle into a final, stable state, however once $\partial U2done$ occurs there

is another 25ns of dead time before $U2$ is triggered again. Therefore the cycle time for $U2$ could be safely increased by up to 25ns, and by providing $\partial U1done$ as the ∂out signal, a 10ns improvement is achieved. The activation of $U2$ is now performed *in parallel* with the handshaking and control logic of the environment. Fig.4.5 shows the resulting ECS system, with an improved cycle time of $\sigma = 25ns$ (a 31% improvement). Note also that the control logic within $U2$ and $U3$ is simplified since there is now no need to generate a *done* event.

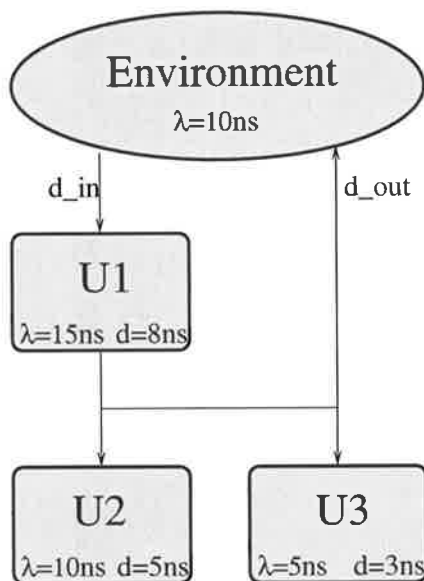


Figure 4.5: An improved ECS version of the example system.

For this scenario it is not possible to take this approach to its extreme and issue ∂in directly as the returning acknowledgement ∂out to the environment (although in other situations this *may* be possible). This is because the latency of $U1$ is 15ns but a new input event can occur after only 10ns, therefore the first operation could easily become corrupted by the next.

It may however be possible to improve the *internal* latency of $U1$ to just greater than $d = 8ns$. This is because at present $U1$ is still implemented as a SI unit, with ∂out being issued once all internal states have settled. However, there exists a 10ns dead time before its next activation (due to the environment's latency), and so ∂out can safely be provided earlier by $U1$, whilst its internal states settle in parallel with the handshaking and control logic of the environment. For example, assuming an internal event from $U1$ can be used as ∂out after 8ns (to coincide with the data output), then the cycle time of the new system will be reduced to just $\sigma = 18ns$ (a 50% improvement on the original SI implementation).

4.2.1 Example: sharing of a common unit

As a simple example of the improvement which can be attained by removing acknowledgements, consider the circuit shown in Fig.4.6a, which shows one way in which a computational block X can be shared by two processes.

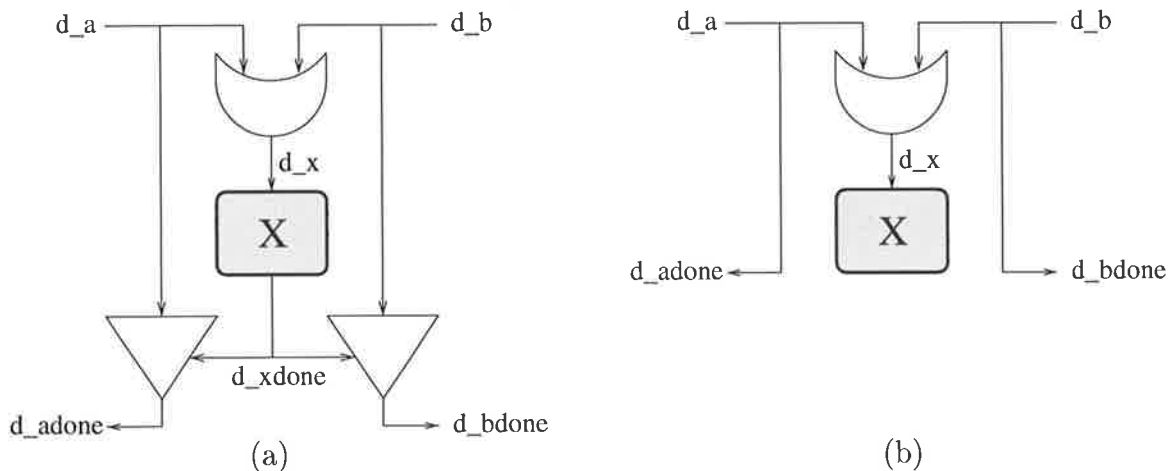


Figure 4.6: (a) A SI and (b) an ECS circuit for sharing a common unit.

For the SI circuit $\partial adone$ and $\partial bdone$ cannot be generated until $\partial xdone$ has occurred, however no such restriction is present in ECS. If it is known that the data produced from X does not interact with the subsequent event control of either $\partial adone$ or $\partial bdone$ for at least $t_{xor} + t_X$, where t_{xor} and t_X are the processing delays of the xor gate and block X respectively (a convention regularly adopted throughout this thesis), then ∂a and ∂b can in fact feed out directly as the *done* signals.

The ECS implementation is shown in Fig.4.6b. As a result of removing the acknowledgement from X , the latency of the design has been improved by $t_{xor} + t_X + t_{restore}$, and the complexity of X has been reduced by the redundancy of $\partial xdone$ and the shift in paradigm from SI to ECS. The control area has also been reduced from three gates to one, which in turn results in a reduction in power dissipation (from two gates having to switch per cycle down to just one).

This type of optimization is especially applicable when X has a low processing delay, since there is then a greater probability of the data being used by the *done* circuitry after $t_{xor} + t_X$.

4.2.2 Example: data latching circuits

As a more comprehensive example of the speed improvements that can be achieved through the removal and reduction of acknowledgements, and also to further expound the advantages of ECS over SI designs, consider now the specification for a data latching circuit (DLC). An incoming event ∂in is required to latch the data present on the *input* bus to the *output* bus, and then issue ∂out when complete.

The simplest and fastest SI approach utilizes the event-driven latches (as shown in Fig.3.11) of the micropipeline methodology [Sut89]. This is shown with inverter drivers in Fig.4.7a.

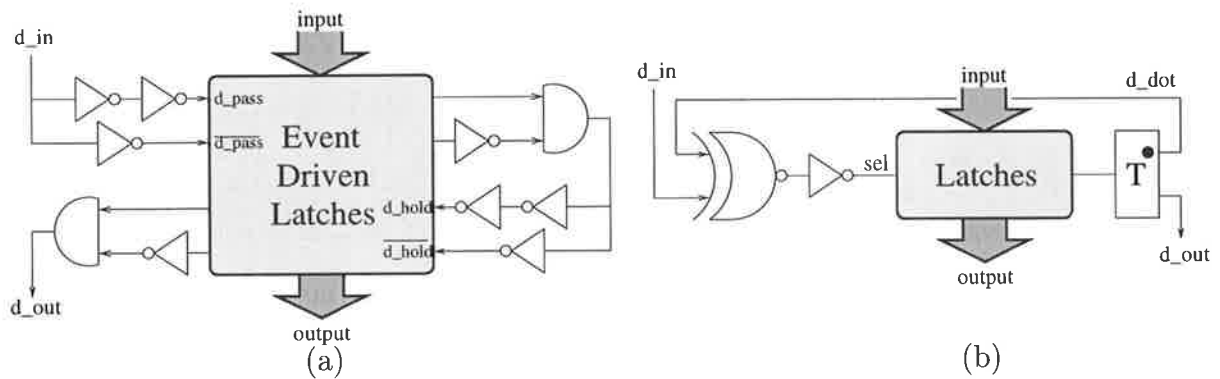


Figure 4.7: Two SI micropipeline control structures for the DLC using (a) event-driven latches, and (b) logic-driven latches.

This is perhaps the fastest SI implementation of a data latching circuit, with a latency of $\lambda = 4t_{inv} + 2t_{wire} + 2t_{cgate}$, where t_{wire} represents the propagation delay of the signal through the latches. The problem however is that each event-driven latch requires at best 18 transistors, which makes for a rather large implementation of the design for all but the smallest of data widths. A more economical solution is to use the logic-driven latches of Fig.3.5a, and use a combination of *xor* and *toggle* to generate the select line pulse from the incoming ∂in event. This implementation is shown in Fig.4.7b.

The *toggle* operates such that the first event to occur on its input (Δsel , as generated by the *xor* of ∂in and ∂dot) is steered to the *dot* output, and subsequent input transitions oscillate between this and ∂out . The occurrence of ∂dot causes ∇sel to occur, which is then steered to ∂out via the *toggle* to indicate completion of the latching cycle. This design requires only 6 transistors per latch and is therefore a much smaller implementation than the event-driven latch design. Furthermore, its speed is still comparable to that of

Fig.4.7a, with a latency of $\lambda = 2(t_{xor} + t_{inv} + t_{wire} + t_{toggle})$.

In ECS no toggle element exists, however its functionality can be mimicked by using two send gates driven by complimentary control signals, as shown in Fig.4.8.

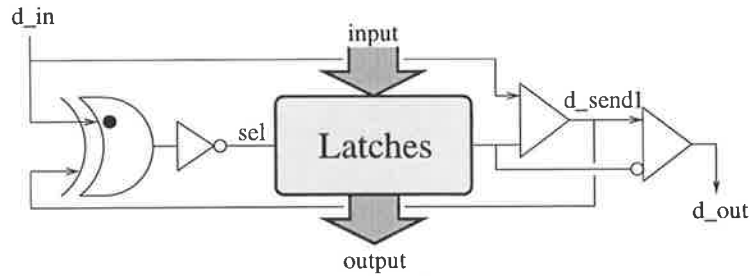


Figure 4.8: An ECS implemented SI version of the DLC.

In this instance the incoming event ∂in is passed through three phases: the first raises the select line, the second lowers the select line (once Δsel is detected at the first send gate), and the third issues ∂out (once ∇sel is detected at the second send gate). This implementation is known as a *split toggle*, and results in a SI implementation with a latency of $\lambda = 2(t_{xor} + t_{wire} + t_{send}) + 3t_{inv}$.

Although the above three implementations operate using SI control, it *must* however be noted that this still does not guarantee that the data has been latched! To do this would require comparing *input* with *output* for ALL bits and enabling ∂out (through a *send* gate) when they are all identical. This in practice would severely complicate the control schema (in terms of both speed and area) and is therefore rarely ever implemented. However without this, although the control schema alone may satisfy SI criteria, the data path does not, but satisfies instead a BD model. Therefore despite claims to SI designs [BS89, SMJ⁺94, ABV⁺95], such as those given above, the reality is that the overall system is in fact a BD implementation!!

Given the circuit of Fig.4.8, there are still a number of improvements which can be made to improve the cycle time by removing BD redundancies and making assumptions about the environments operation. The most obvious improvement is to use $\partial send1$ as ∂out , and so remove the delay imposed by waiting for the circuit to settle to its final state. This implementation assumes that ∇sel can occur in parallel with the control circuitry of the environment, and imposes a limit upon the minimum re-activation time of the circuit (from ∂out to the appearance of new data on *input*) of $t_{xor} + t_{inv} + t_{wire}$.

Furthermore, it is usually possible to ignore the wiring delay through the latches, since

this is typically very small in comparison to the gate delays (except for very long data bus widths, which would be extensively buffered anyway), and also because the Δsel propagation delay is not problematic to the circuit unless the ∇sel delay is significantly less (which would destroy the select pulse for the furthest latches), and this rare occurrence can be easily solved through alternative transistor sizing. As a consequence, the ECS circuit implementation of Fig.4.9a results, which is basically identical to the self-timed pulse circuit shown shaded in Fig.4.1.

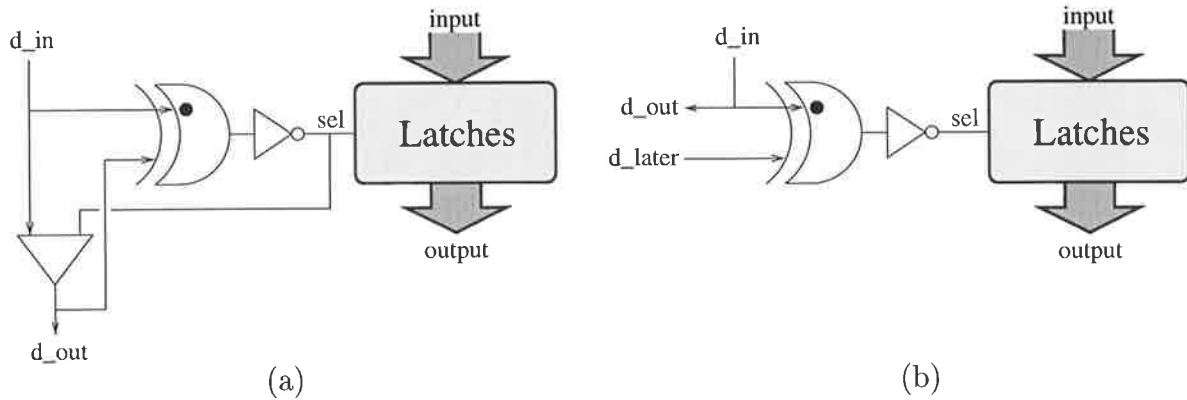


Figure 4.9: (a) A semi-optimized and (b) a fully optimized ECS implementation of the DLC.

By ignoring the wiring delay, all of the event control can now be placed on the same side of the latching array. This enables a better VLSI floorplan and implementation of the circuit, since only one control block must be designed and is effectively independent of the data path. The latency of this ECS implementation is only $\lambda = t_{xor} + t_{inv} + t_{send}$.

In some instance the speed of this circuit can be improved even further. If it is known that within the environment the data on *output* does not interact with the subsequent event control of ∂out for at least $t_{xor} + t_{inv} + t_{latch}$ (an identical consideration to that discussed in the previous section), then the event ∂in can in fact be fed out *directly* as ∂out , resulting in a zero latency implementation of the DLC.

In fact, if within this subsequent event control an event $\partial later$ is generated, then the *send* gate can be removed and $\partial later$ fed back to the DLC to activate ∇sel . This also requires that the delay between ∂in and $\partial later$ is at least as long as the pulse width necessary for the latching of the data: $\partial in \rightarrow \partial later > t_{latch}$. This circuit optimization is shown in Fig.4.9b, and although it does not (and cannot) further reduce the latency of the design, it does reduce the area of the control circuitry by a *send* gate.

Table 4.2 shows a comparison of the DLC implementations in terms of the gate delays

and relative speeds, using the results of Table 3.1 and assuming a relative *toggle* delay of 13 [Pav94, page 53], and $t_{wire} = 0$.

Circuit Implementation	Gate Delay Path (λ)	Delay
Micropipeline with event latches (SI)	$2(2t_{inv} + t_{wire} + t_{cgate})$	28
Micropipeline with data latches (SI)	$2(t_{xor} + t_{inv} + t_{wire} + t_{toggle})$	44
ECS using a split toggle (SI)	$2(t_{xor} + t_{wire} + t_{send}) + 3t_{inv}$	32
ECS with initial optimization	$t_{xor} + t_{inv} + t_{send}$	15
ECS assuming a ∂ later event	0	0

Table 4.2: Relative speeds of five different implementations of the DLC.

It is evidenced that even the least optimized ECS design results in over a 46% improvement in latency over the fastest SI implementation, as well as considerably reducing the circuit area and control complexity. At best, the ECS methodology results in the minimum possible latency of 0ns and the minimum control area of just an *xor* and a *driver*. Furthermore, for this circuit the number of gates switching per cycle has been reduced from six in Fig.4.7b down to four in Fig.4.9b, which also indicates a reduction in power dissipation.

4.2.3 Comments on improving acknowledgements

Providing earlier acknowledgements, or removing unnecessary ones, has been shown to result in significantly faster system speeds in ECS than can be achieved using a SI implementation, however to use these techniques also requires some knowledge of the environment’s operation. In particular, the time taken before the results of the optimized circuit are again utilized, and the delay time between successive activations. Typically, even at the architectural level, these time frames are reasonably well known (at the least, a minimum delay can be determined), and as such these techniques can be regularly applied.

These optimizations can be applied successively to each sub-component of a system (beginning with the most critical), and if possible verifying through high-level simulation that the system’s functionality is still correct. As new components of the system are designed, a greater knowledge of their environment will become available, and so the optimization techniques can be re-applied with progressively better results, and recursively applied to previously designed components to further improve system speeds.

4.3 Activating functional units

It is often the case that two or more functional units (FUs) are required to be activated (perhaps conditionally) by an incoming event, and to produce a completion event which covers all possible activation scenarios. This problem can be somewhat complex (especially for a large number of FUs and controlling conditions), and requires the use of advanced techniques to provide for low latency solutions.

4.3.1 Conditionally activated parallel units

Figure 4.10 shows a typical situation in which under all circumstances unit X is activated, however unit Y is only activated (in parallel with X) when a governing signal c is high. The implementation of this is obvious, however the problem posed is how best to generate a ∂out event for this system?

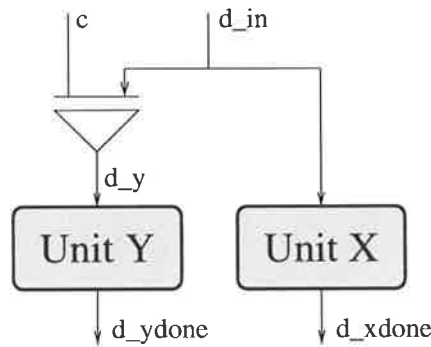


Figure 4.10: Conditional activation of unit Y in parallel with unit X .

One approach would be to devise a control circuit based on the *logical* necessities for ∂out : **if** $c = 0$ **then** $\partial out \leftarrow \partial xdone$ **else** (with $c = 1$) $\partial out \leftarrow \partial xdone.\partial ydone$. Each of the expressions in this logical description can be translated into an ECS gate. The process of selection (*if ... else ...*) translates into a *select* gate, the assignment when $c = 1$ is clearly a *gate*, and the assignment to ∂out under *both* clauses of the selection statement translates into a *merge* gate (an OR function to events) to produce a ∂out event under both exclusive conditions.

This control circuit is shown in Fig.4.11a, and is in fact a SI implementation of the design. Its functionality is clearly as required by the logical description given earlier, with ∂out being triggered by $\partial xdone$ when Y isn't activated (for $c = 0$), and by the *and* of $\partial xdone$ and $\partial ydone$ when it is (for $c = 1$). The best case latency for this circuit is

$\lambda_{best} = t_{cgate} + t_{merge}$, and the worst case latency is $\lambda_{worst} = t_{feed} + t_{cgate} + t_{merge}$, given that the *select* and *feed* gates have identical VLSI implementations (see Section 3.2.4).

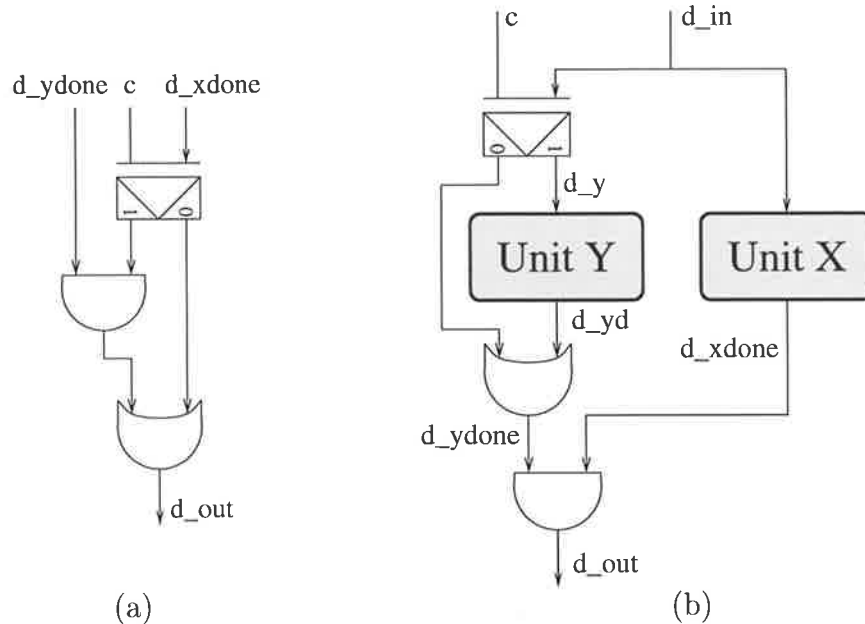


Figure 4.11: (a) A SI implementation for generating a ∂out event, and (b) an improved SI implementation.

Note that $\partial xdone$ is used in all circumstances in the generation of ∂out , and it therefore seems inefficient to have to split this signal via the *select* gate only to merge it again later. In fact, it can be observed that if a $\partial ydone$ signal could be produced for both $c = 0$ and $c = 1$, then ∂out would be given simply by: $\partial out \leftarrow \partial xdone.\partial ydone$.

The logical requirements for generating $\partial ydone$ under these conditions can be specified as: **if** $c = 0$ **then** $\partial ydone \leftarrow \partial in$ **else** $\partial ydone \leftarrow \partial yd : \partial y : \partial in$, where ∂yd is now the done signal provided from unit Y . As before, the selection clause of this statement translates into a *select* gate, and the two separate assignments to $\partial ydone$ translates into a *merge* gate. Fig.4.11b shows the resulting circuit for this implementation, which is still speed independent.

It will be observed that whereas in Fig.4.10 only a *feed* gate was used to trigger unit Y , this has now been replaced by a *select* gate. This effectively generates a bypass event to $\partial ydone$ when unit Y isn't activated, and ensures that regardless of the state of c , an output event is always produced.

By viewing the design problem as a whole, and not as two separate components for the input and output circuitry, a faster and smaller implementation has been produced. The best case latency is now $\lambda_{best} = t_{cgate}$, and the worst case latency is $\lambda_{worst} = t_{merge} + t_{cgate}$,

both of which are an improvement on the completion generation of Fig.4.11a.

An ECS technique can be used which reduces the latency even further. As already mentioned, ∂x_{done} is always used in the generation of ∂out , and so a *send* gate can be used to effect this. The governing control signal can be given by whether or not unit *Y* is active, since if it is, then ∂x_{done} must be kept pending (if it has occurred) until unit *Y* becomes inactive, otherwise it can pass through immediately as ∂out . This control circuit is shown in Fig.4.12.

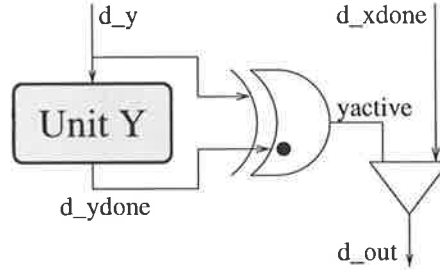


Figure 4.12: An ECS implementation for generating a ∂out event.

This design assumes a minimum operating time for unit *X* which enables *yactive* to be set: $t_X > t_{feed} + t_{xor}$, and is usually valid for all but the smallest of circuits. This ECS implementation results in a faster generation of ∂out than can be achieved with the fastest SI design. Table 4.3 summarizes the completion latency (from $\partial done \rightarrow \partial out$) for when unit *X* alone is triggered, and for when both units are triggered with their $\partial done$ events occurring to give the worst case latency, again assuming the relative gate delays quoted in Table 3.1.

Circuit Implementation	Only unit <i>X</i> triggered		Both units: worst case	
	Gate Delay Path	Delay	Gate Delay Path	Delay
Initial SI	$t_{feed} + t_{merge}$	20	$t_{feed} + t_{cgate} + t_{merge}$	30
Improved SI	t_{cgate}	10	$t_{merge} + t_{cgate}$	17
ECS	t_{send}	6	$t_{send} + t_{xor}$	13

Table 4.3: Relative speeds of three different implementations of a conditional triggering circuit.

Yet again, the ECS implementation has surpassed the best SI design by a significant factor (up to 40% in this instance).

4.3.2 Generating a ∂out event in the general sense

The technique of the previous section can be extended to the general case in which *p* units are always triggered when a ∂in event occurs, and *q* units are conditionally triggered

according to q separate, but not exclusive, control signals. This situation is shown in Fig.4.13a, which indicates how a SI generation of ∂out is performed as extended from Fig.4.11b. Figure 4.13b shows the corresponding ECS implementation as extended from Fig.4.12.

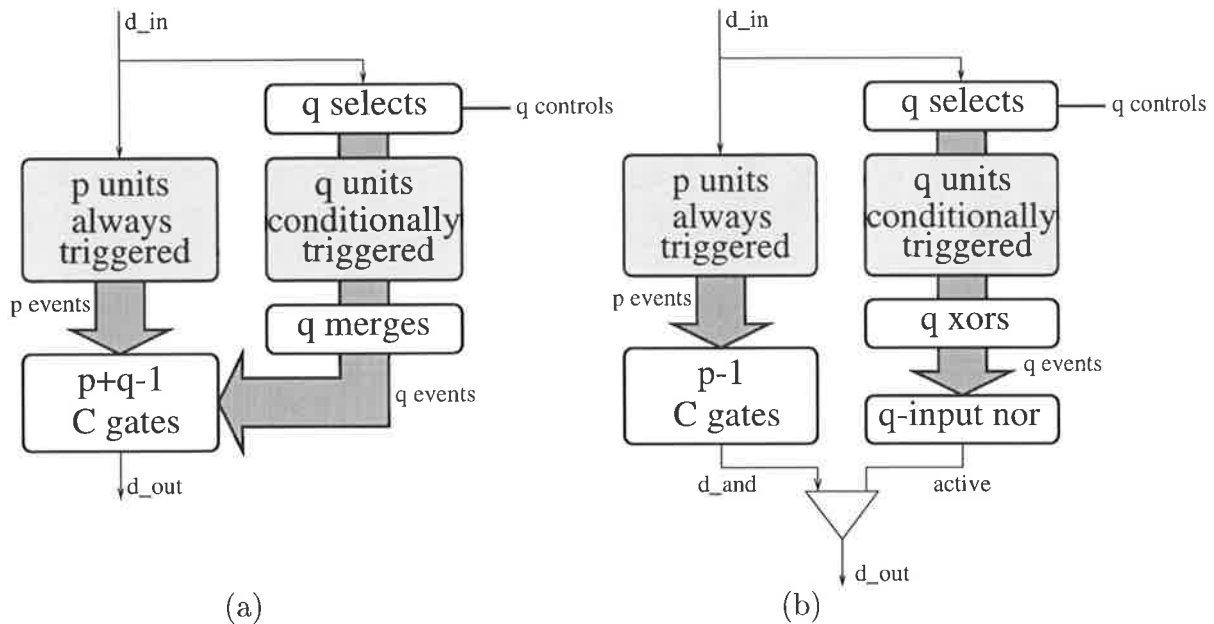


Figure 4.13: A generalized conditional trigger structure with (a) a SI implementation for generating ∂out , and (b) an ECS implementation.

Table 4.4 indicates the best and worst case latencies for the SI and ECS implementations, assuming that the $(p + q)$ -input *cgate* is implemented as a binary tree, and that the q -input *nor* gate is implemented as a pseudo-nmos structure (with a single pull-up transistor tied to ground), with an estimated relative delay of $t_{pnor} \gtrsim t_{inv} = 3$. Note that for $p \leq 1$, the best and worst case latencies for the ECS implementation are swapped, the floor ($\lfloor \cdot \rfloor$) and ceiling ($\lceil \cdot \rceil$) functions round to the nearest lower and upper integers respectively, and the delay values are quoted for the specific case of $p = 4$ and $q = 5$. For larger values of p and q , the relative improvement of the ECS approach increases (and conversely for smaller values).

Circuit Implementation	Best case ($p > 1$)		Worst case ($p > 1$)	
	Gate Delay Path	Delay	Gate Delay Path	Delay
SI	$\lfloor \log_2(p + q) \rfloor \cdot t_{cgate}$	30	$\lceil \log_2(p + q) \rceil \cdot t_{cgate} + t_{merge}$	47
ECS	$t_{xor} + t_{pnor} + t_{send}$	16	$\lceil \log_2(p) \rceil \cdot t_{cgate} + t_{send}$	26

Table 4.4: Relative speeds of two different implementations of a generalized conditional triggering circuit.

The ECS implementation has in this instance resulted in a 47% improvement over the SI design. The only constraint on this circuit is that $T_{P_{min}} > t_{feed} + t_{xor} + t_{pnor} - \lfloor \log_2(p) \rfloor \cdot t_{cgate}$, where $T_{P_{min}}$ is the minimum latency of the fastest p unit. This ensures that ∇_{active} has occurred (if at all) before ∂_{and} .

4.3.3 Generating a ∂_{out} event for exclusively triggered units

As a frequently occurring variation on this theme, consider now the case in which only one of r functional units can be exclusively triggered by the ∂_{in} event. It is possible to use the generalized SI circuit above (with $p = 0$ and $q = r$), but a faster SI implementation results if a tree of select gates is used to conditionally trigger each unit, and their outputs merged together to generate ∂_{out} (instead of *anding* a merge from each gate as before). This is shown in Fig.4.14a.

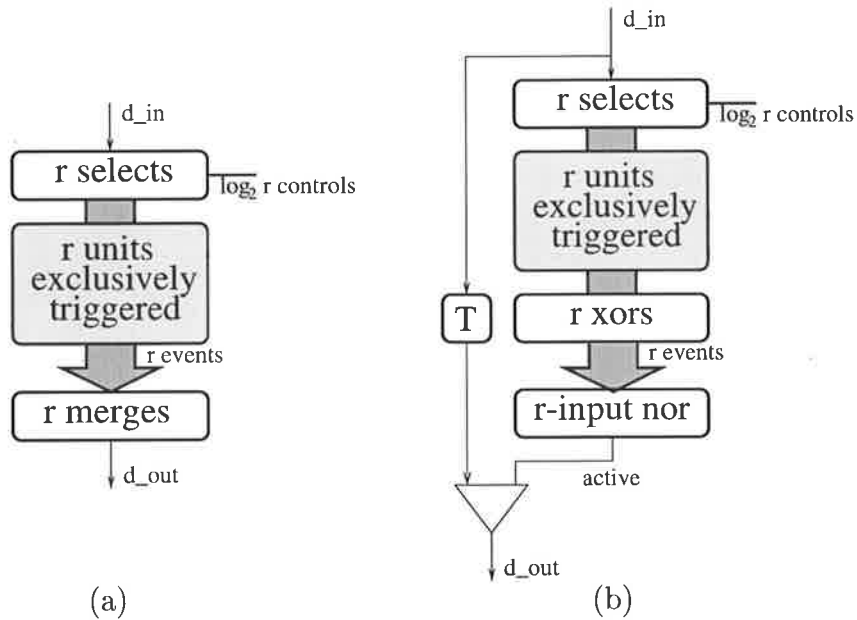


Figure 4.14: Generating ∂_{out} for exclusively triggered units in (a) SI and (b) ECS.

A similar ECS design to that of the previous section can also be employed, with the only difference being that instead of triggering the *send* gate from a tree of *cgate*s, a delay unit T is employed which simply delays the input event until the *active* signal is set. This places no constraints on the operating speed of any of the units, but instead places a design constraint on the delay: $T > \lfloor \log_2(r) \rfloor \cdot t_{feed} + t_{xor} + t_{pnor}$. This implementation is shown in Fig.4.14b.

The average completion latency of the SI design (from $\partial rdone_{last} \rightarrow \partial out$) is given by $\lambda_{ave} \approx (\log_2 r) \cdot t_{merge}$ and that of the ECS design is $\lambda_{ave} = t_{xor} + t_{pnr} + t_{send}$, which can be shown to be faster than the SI implementation for $r > 4$, assuming the relative gate delays of Table 3.1.

4.3.4 Splitting a tree of *select* gates into individual *feed* gates

The previous section referred to a tree of *select* gates which was used to exclusively trigger one of r FUs. Although this keeps the control logic for the data path simple, the event path latency to trigger can become quite large: $\lambda_{worst} = \lceil \log_2(r) \rceil \cdot t_{feed}$. A better solution is to generate a selection signal for each separate FU (perhaps with a decoder), and individually trigger each FU through a *feed* gate. Although this approach complicates the data path, the event latency is significantly improved to a constant $\lambda_{worst} = t_{feed}$. This would also improve on the design constraint for the delay T , to simply: $T > t_{feed} + t_{xor} + t_{pnr}$.

4.4 Reducing event path delays

All of the previous sections have provided general techniques for improving system speeds by typically reducing the complexity of gates in the critical event path. There are still other techniques which can be applied in more application specific circumstances to further reduce the event path delays of a circuit.

4.4.1 Moving metastability detection out of the event path

Consider now the situation in which a signal is used to halt an event (as per a *send* gate), however in this instance the data and event are not *associated*. That is, a negative transition of this signal may occur at any time and is uncorrelated to the occurrence of the event. Clearly then, an implementation involving just a *send* gate is unacceptable, since the output may become metastable when the input event and negative transition of the control signal occur in such close proximity that the output may hover indefinitely between the new and old logic levels.

To ensure that a *valid* logic level is seen by the subsequent control circuitry, a metastability resolver (MR) must be placed after the *send* gate as shown in Fig.4.15a. The

MR must ensure that a transition on ∂out is not activated until $\partial send$ is out of the metastable region. This could be implemented by a VLSI circuit which doesn't trigger a logic high transition on its output until its input has exceeded some high voltage threshold VHT , which is outside of the metastability region (which itself is typically near $VDD/2$ [CM73]). Similar arguments apply for triggering the logic low transition. For example, with $VDD = 5V$, one may design the MR for $VHT = 3.5V$ and $VLT = 1.5V$.

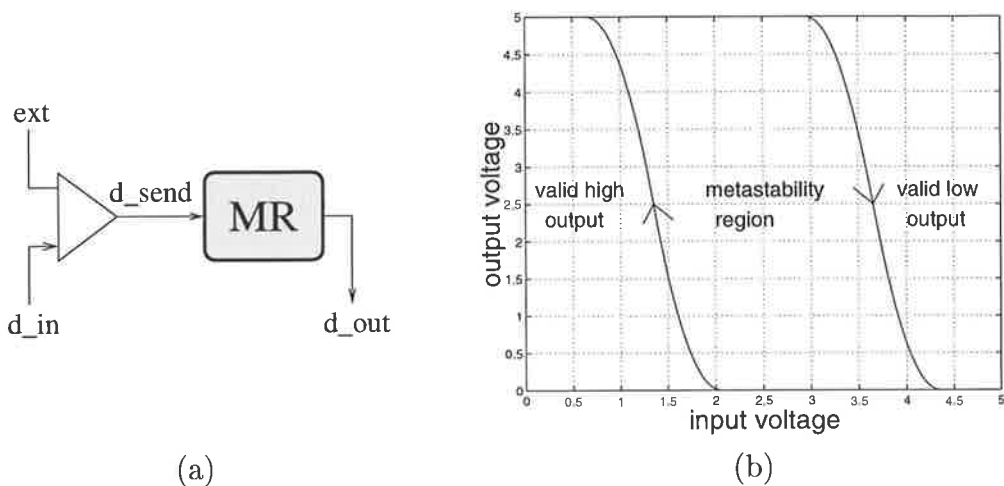


Figure 4.15: (a) An implementation of the unassociated halting circuit, and (b) the transfer characteristic of its MR.

The important point to note about such a circuit is that a *hysteresis* loop is present, as described by the transfer characteristic of Fig.4.15b. This requires a rather complex design (such as a Schmitt trigger [GD85]) which can significantly increase the latency from ∂in to ∂out for the case when the signal ext remains high. In many instances, this may in fact be by far the most frequent scenario (such as for interrupt or exception processing), and it is therefore worth investigating alternative architectures which reduce the *typical case* latency.

One such approach is illustrated by Fig.4.16a, which operates as follows. The incoming event ∂in places the event driven latch into the *pass* state, which enables the ext signal to propagate through to $extmeta$ (although its inverted output $\overline{extmeta}$ is actually used). Concurrently, based on the previous value of $extok$ which is here assumed to be high, the event ∂in is sent through to ∂out , which then forces the latch back into the *hold* state to retain the latched value of ext (the delay element is used merely to ensure a sufficiently long latching time of $t_{send} + T$).

It will be observed that the signal $\overline{extmeta}$ now has the potential for metastability,

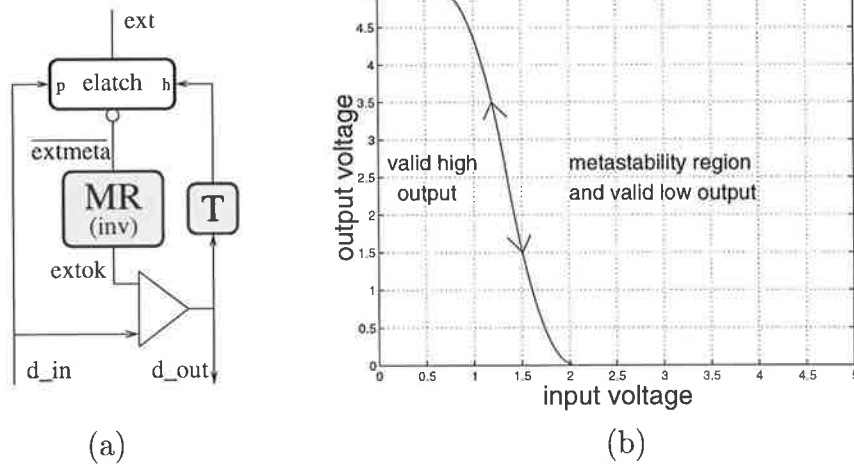


Figure 4.16: (a) An improved implementation of the unassociated halting circuit, and (b) the transfer characteristic of its MR.

and so a MR is needed after this to ensure that $extok$ presents a valid signal to the $send$ gate. This MR has a simpler requirement than the previous one, since in this instance a problem only exists for when $\overline{extmeta}$ is transitioning high (a low transition implies that din is already stalled at the $send$ gate), whereas previously *both* transitions of the $\partial send$ signal had to be resolved. The transfer characteristic for this MR is shown in Fig.4.16b, and it can be seen that no hysteresis is present. Therefore the MR can be implemented simply as an inverter with a low switching threshold (say, $VTL = 1.5V$), which requires nothing more than a high $n:p$ transistor width ratio and is a significantly less complex implementation than required previously!

The important issue however with regards to this circuit is that, for the typical case when $extok = 1$, the latency of the circuit is merely $\lambda = t_{send}$. There is however a requirement on the environment, to ensure that a newly latched value of $\overline{extmeta}$ reaches VTL in the worst case (causing $\nabla extok$) before the next din event occurs: $\partial out \rightarrow \partial in > \Delta_{VTL} t_{elatch} + \nabla t_{inv(MR)} - t_{send}$. This constraint in practice imposes a negligible minimum latency on the environment, which is usually obeyed without any additional design effort. A requirement on the circuit is that ∂out occurs before $\nabla extok$ in the same cycle, implying: $t_{send} < \Delta_{VTL} t_{elatch} + \nabla t_{inv(MR)}$, which is also usually obeyed without any extra design effort. Note also that the initial state of the inverting latch must be set (typically low).

Consider now the other case in which din occurs when $extok$ is low, either due to continuing metastability of $\overline{extmeta}$ from the previous cycle, or a valid logic high of this

signal. In this instance the output ∂out is simply stalled until such time as ext goes high again, as this then propagates through to $extok$ since the latch is in the pass state.

An Hspice simulation of the new architecture is shown in Fig.4.17a for a sweep of times for ∇ext through the metastability region and an $n:p$ width ratio of the MR inverter of 15:1. This was implemented in VLSI using the $0.7\mu m$ DLM CMOS ES2 technology, with a 5V supply, a temperature of $75^\circ C$, and using “typical” process parameters (hereafter simply referred to as the ES2 technology).

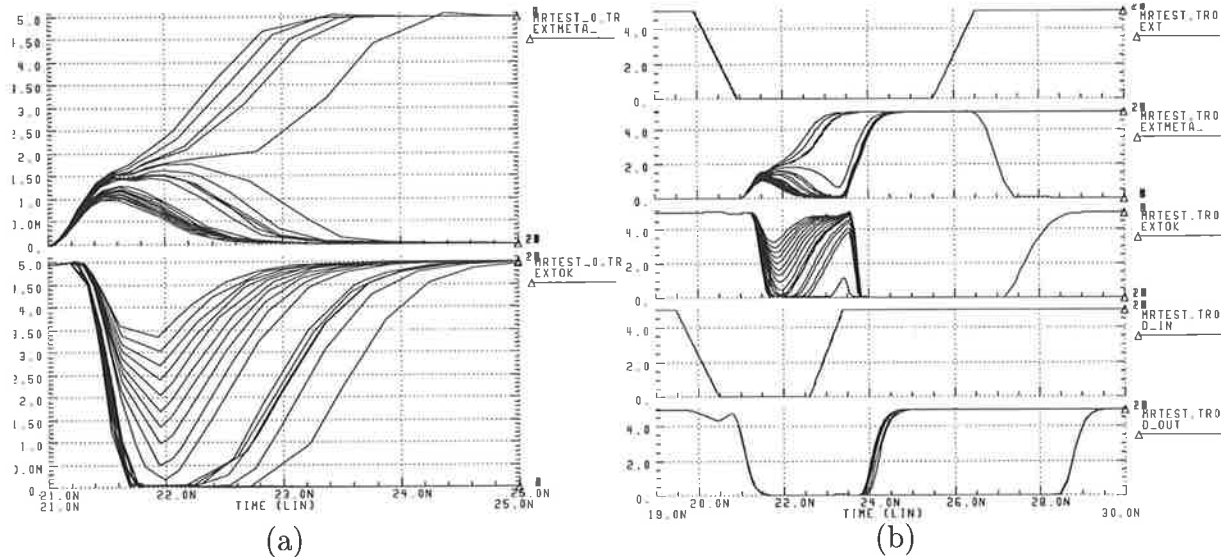


Figure 4.17: (a) An Hspice simulation around the metastability region of the improved halting circuit, and (b) the circuit’s operation once the next ∂in event is applied.

Around the metastability region of $\overline{extmeta}$ (at about 1.9V for this process), the signal $extok$ has safely been set low, and therefore metastability can exist indefinitely without causing a problem for the $send$ gate. The potential danger now however is when $\overline{extmeta}$ rises to a peak of around 1.1V, which causes $extok$ to dip to around 2V and may then cause a problem for the $send$ gate. However, being outside the metastable region, it is known that $\overline{extmeta}$ will fall again and hence $extok$ will rise. Therefore although $extok$ dips it is *not* metastable, and as can be seen from the simulation, is in all instances always rising after $t = 22ns$. The next ∂in event can safely be applied to the $send$ gate after this time (since being concurrent with a rising edge of the control signal is not a hazard) as shown in Fig.4.17b (with ∂in occurring at $t = 23ns$). Since the latch is now in the pass state, the low value of ext is sent through to $extok$. If it was already low, then no ∂out event will occur until Δext occurs (ie- the interrupt has been effective on this cycle, as shown by the ∂out events at $t = 29ns$), otherwise if it was high then a ∂out event will

occur at once (at $t = 24ns$) and the interrupt will be effective on the next ∂in event.

Given that ∂out occurs at about $t = 21ns$, the minimum latency constraint on the environment is therefore $1.0ns$, which is of the order of a typical gate delay and would invariably be met by the environment's circuitry (requiring no extra design effort). The circuit constraint explained earlier has also been met without effort.

By moving the control complexity out of the critical event path and into the data path, the implementation of the MR has been drastically simplified, and more importantly, the typical case latency has been improved.

4.5 Summary

The most recurring theme in all of the techniques presented in this chapter is the minimization of gates in the critical event path. This can be achieved through the basic principles of algebraic reduction, issuing earlier acknowledgements, and transferring complexity from the event path into the data path.

It is clear then that the ECS designer needs to earnestly investigate his or her initial implementation of a circuit (which should first be verified through simulation as functional) to ascertain where it is possible to apply these optimizations and produce a faster (and often smaller) design. As has been shown by the examples given, significant speed improvements of the order of 50% can often be achieved through the use of these techniques. Furthermore, an area reduction has also resulted in the majority of cases which in turn implies fewer gate transitions per cycle and a reduction in power dissipation.

Chapter 5

Asynchronous Pipelines

PIPELINING is an important aspect of systems design and is employed in almost every modern commercial processor. By partitioning a computation into smaller and faster components (such as a 32 bit addition into 32 one bit adder cells) each can be made to function concurrently but on different operations. When one component finishes its part of the computation, it latches its results into the next stage (which does the next part of the computation) and then repeats this process for the next operation. By pipelining a design in this fashion, the throughput can be made to be as fast as any one component rather than their sum, which often results in a substantial improvement in cycle time. This is however at the cost of an increase in latency (due at the least to the propagation delay of the latches between stages), although for many applications this is of less importance than the need for achieving a high throughput. The importance of pipelining is therefore critical to the design of both synchronous and asynchronous systems, of which the latter forms the focus of this chapter.

In the synchronous domain pipelining is effected by subdividing the computational requirements into blocks of approximately equal delays, and then clocking the results of one block into another with a clock period at least as great as the longest delay. Ideally there is no additional control overhead apart from that imposed by this minimum clock period, however in practice a number of other constraints such as skewing, power dissipation, and a large clock driver load all serve to complicate the global clock paradigm (see Section 1.1.1). Furthermore, it is not possible to take advantage of data dependent computation times, and the throughput and latency are governed by the computation time of the *slowest* stage (hence the need to equalize stage delays as much as possible).

Asynchronous pipelines employ the same partitioning principle as a synchronous pipeline, however the requirement of equalized stage delays is relaxed and the regulation of data between blocks is locally controlled. Typically a request-acknowledge (*req-ack*) protocol is employed, such that one stage signals to the next when its data is ready (*req*), and that stage signals back (*ack*) to indicate when its next operation may begin. A variation on this theme, in which no acknowledgements are necessary and timing constraints are propagated back to the input stage, is proposed in [AML96].

A number of different pipeline structures have been designed by various researchers based on the *req-ack* protocol using both two and four phase SI and BD models. The following sections discuss some of these designs and illustrate their comparative performance. Furthermore, some 2P ECS pipeline structures are presented which provide a faster throughput than any others previously reported.

5.1 FIFO pipelines

A FIFO (first-in, first-out) pipeline is one which employs no processing of data between stages, therefore the only limitation on the pipeline's latency is the propagation delay of data through the latches (t_{latch}). FIFOs are often used as storage buffers to a circuit which may exhibit occasionally long cycle times (such as writing to a cache or buffering prefetch logic [LCT⁺95]), so that these variations do not typically impinge upon the cycle time of the source which continues to supply the buffer during these long cycles.

5.1.1 Micropipeline 2P FIFOs

Perhaps the most famous implementation of an asynchronous FIFO is the *micropipeline* proposed by Sutherland [Sut89], as shown in Fig.2.2 of Section 2.2.1 (with processing). One stage of the micropipeline control using logic driven latches is shown in Fig.5.1.

The micropipeline utilizes a 2P SI control schema and operates as follows. Initially the select line *sel* is high so that all of the latches are transparent (enabling input data to filter through), and the *cgate* is primed. An incoming $\partial reqin$ event will therefore propagate through the *cgate* and force the select line low, thereby latching the input data (which must be valid prior to this). The ∇sel transition then proceeds through the *toggle* and emerges as an event from the *dot* output $\partial reqout$, which initiates the operation of

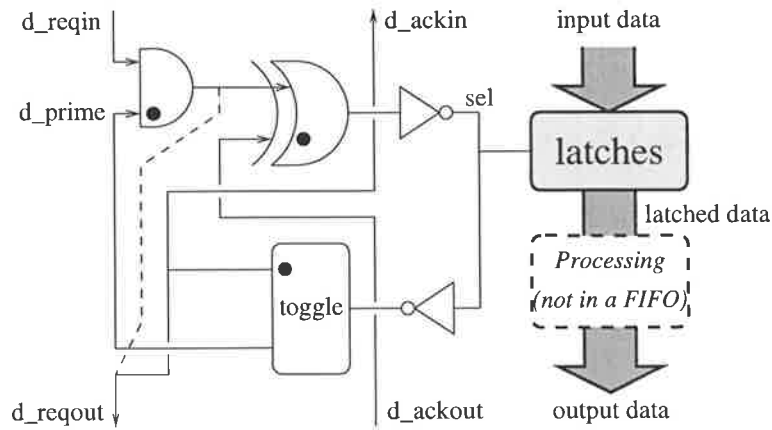


Figure 5.1: A micropipeline stage also indicating a *fast-forward* (dashed line) implementation.

the next stage. This same event is also fed back to the preceding stage as $\partial ackin$ to indicate that the data has been latched and that new data may now be supplied. At some later time the following stage will provide a $\partial ackout$ event to indicate this same situation (that new data may be supplied), which then causes sel to go high via the xor and hence enables new input data to propagate through the latch. This transition then re-primers the $cgate$ via the $toggle$ (emerging now as $\partial prime$). A new cycle begins when the next $\partial reqin$ event occurs, or if one is already pending at the input of the $cgate$. Note that if a latch structure is utilized which requires both sel and its inverse, then a SI control implementation would require a primed $cgate$ to join their opposing transitions prior to the $toggle$ [Sut89, Fur96], however the latch structure of Fig.3.5a avoids this additional overhead since it only requires the sel signal.

One of the problems associated with the micropipeline is the long stage latency from $\partial reqin$ to $\partial reqout$ ($t_{cgate} + t_{xor} + 2t_{inv} + t_{toggle}$), which is significantly greater than the minimum allowable latency of t_{latch} . As an improvement, the dashed event of Fig.5.1 can be used to provide an early $\partial reqout$ event to the next stage. This improves the stage latency to just t_{cgate} (which is still slightly greater than the latch propagation delay) and also subsequently improves the cycle time. Note however that this alternative control implementation (dubbed a fast-forward micropipeline [Sut89]) is no longer SI.

5.1.2 4P FIFO circuits

A very simple 4P control circuit [FES94] is shown in Fig.5.2, and consists merely of a $cgate$. Much of the control circuitry of Fig.5.1 has been made redundant since a 4P

signalling protocol has a RTZ phase, thereby removing the need for a two-to-four phase conversion for the *sel* signal (though the *xor* and *toggle*). Instead, the *cgate* can be used directly to drive the latches (through a driver) and to signify the *ackin* and *reqout* signals to adjacent stages.

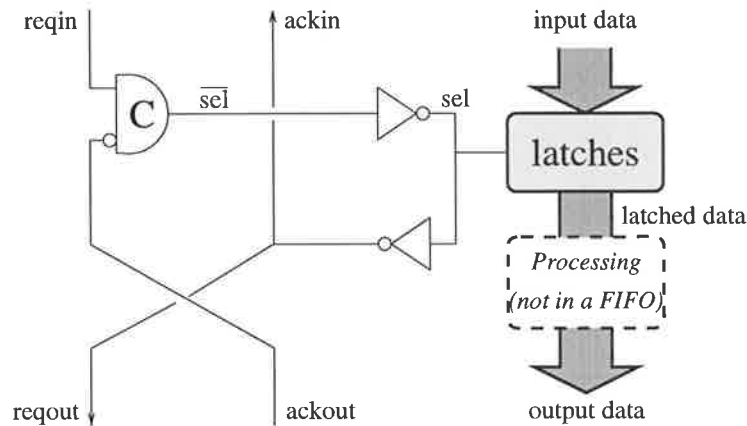


Figure 5.2: A simple four phase (S4P) FIFO controller.

Although this SI control schema is extremely simple it suffers from the fact that it is only ever possible to have alternate stages storing data. This is because the closing of stage i (∇sel_i) causes the opening of the preceding stage (through the path: $\nabla sel_i \rightarrow \Delta ackin_i \rightarrow \nabla sel_{i-1} \rightarrow \Delta sel_{i-1}$) which in turn causes the closing of the one before that (∇sel_{i-2}). Thus, it is only ever possible for every second stage to be latching data (∇sel).

A highly optimized 4P improvement on this implementation is shown in Fig.5.3 [DW95]. This circuit introduces additional control complexity to enable the latch to close before the following stage becomes open (meaning that ∇sel_{i-1} is no longer dependent on Δsel_i), and to also decouple the RTZ handshaking phase with the preceding stage. As such the cycle time and latency of this circuit is improved, despite being a significantly larger design. The asymmetric *cgate* notation used is such that an input striking the + bar affects only the rise of the gate's output (on a positive transition), and one striking the - bar affects only its fall (on a negative transition). Both transitions are controlled by an input striking the body of the gate, as in a conventional *cgate* implementation.

5.1.3 A fast ECS FIFO

It was observed for the micropipeline of Section 5.1.1 that by removing the SI constraint on the circuit (in providing an earlier $\partial reqout$ event) a faster design could be produced. By

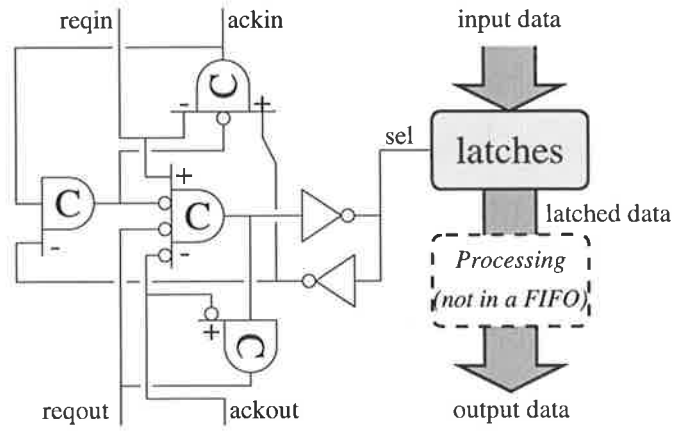


Figure 5.3: A decoupled four phase (D4P) FIFO controller.

taking this ideal one step further, and removing the extra logic necessary for re-converting the *sel* signal back into events (through the *toggle* gate), a much simpler implementation can be produced as shown in Fig.5.4a. In this instance, the output of the *cgate* is used to provide the $\partial reqout$ and $\partial ackin$ events directly. To enable a sufficiently long high signal on *sel* after a $\partial ackout$ event occurs when a $\partial reqin$ event is pending, a small delay must be inserted prior to the priming of the *cgate* (giving a minimum latch pulse width of $T + t_{cgate}$).

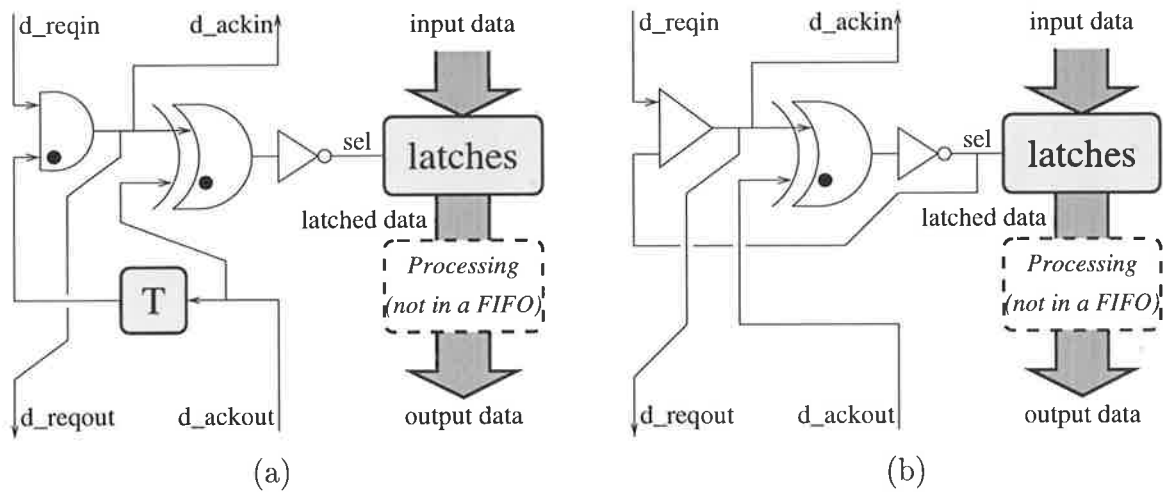


Figure 5.4: (a) An ECS implementation of the micropipeline and (b) the ECS state pipeline.

A similar structure is proposed in [YBA96] which uses very large (24 transistors) double-edge triggered flip-flops in place of the *xor* and logic driven latches (which use only 6 transistors). This results in a substantially bigger design with a slightly slower speed than the circuit of Fig.5.4a (due to the increased load on the select line).

It can be seen that the $\partial ackout$ event of Fig.5.4a results in Δsel , and so to enable a more reliable latch pulse width to be generated, this could be used instead of the $cgate$ structure to produce $\partial ackin$ through a $send$ gate. This implementation is known as the *state pipeline* [MAL95] and is shown in Fig.5.4b. As well as producing a better sel pulse (in a self-timed fashion, since Δsel is now guaranteed to occur before ∇sel), the latency and cycle times of the pipeline are also improved (since $t_{send} < t_{cgate}$ and there is less load on the $\partial ackout$ event).

5.1.4 Comparison of FIFO designs

The micropipeline, four phase, and ECS FIFOs were all simulated in Hspice using the ES2 technology, with four stages implemented and a data width of 32 bits. Identical gate structures and sizes were used between designs which enhanced the accuracy of the comparison. The simulation results are given in Table 5.1, for which the cycle time (σ), latency (λ), power (P), and power-delay ($P\sigma$) values are all quoted *per stage*.

Circuit	σ (ns)	λ (ns)	P (mW)	$P\sigma$ (mWns)
Micropipeline	12.5	4.4	1.8	22.5
F-forward micropipeline	9.3	1.3	2.6	24.2
Simple 4phase	8.1	2.0	1.5	12.2
Decoupled 4phase	7.8	1.6	3.4	26.5
ECS micropipeline	3.5	1.5	3.0	10.5
ECS state pipeline	2.8	1.3	2.2	6.2

Table 5.1: Relative performance of six FIFO circuits.

These figures verify the claim in [DW95] that 4P SI control circuits outperform 2P SI circuits (giving a 38% improvement in cycle time), however the same cannot be said against a BD model. In fact, the ECS state pipeline's cycle time is shown here to be 64% lower (almost 3 times faster) than the best 4P implementation *and* consumes 35% less power! Furthermore, the latency of the state pipeline is also significantly faster than this design (by 35%). It is also interesting to note that the decoupled 4P FIFO is only marginally faster than the single $cgate$ implementation, which also consumes considerably less power (56% less). The power-delay value $P\sigma$ is an often quoted figure of merit which indicates the trade-off between speed and power, and in this category the ECS FIFO is again superior.

It is clear then from these figures that, for a FIFO implementation, a 2P ECS pipeline is significantly faster than a 4P pipeline, and that employing a BD model enables higher circuit speeds to be attained than is possible with a SI model.

5.2 Pipelines with processing delays

The preceding FIFO circuits illustrate the control schemas which can be employed to construct a pipeline, however without incorporating any data processing their applications are limited. Typically, each stage of the pipeline will perform some kind of operation on the data which may take longer than the latency of the FIFO, so the output event to the next stage *ackout* must be stalled until its output data is valid.

In the simplest case for which the data processing delay is bounded and regular (ie does not exhibit significant variations in computation time), the *req* signal can simply be stalled by the use of a delay element, as shown in Fig.5.5. Note however that it is not strictly necessary for *req* to be delayed until its output data is valid. Rather, it must be delayed to the extent that the next stage will activate ∇sel after all valid output data has propagated through the latch. This means that the delay element *T* of Fig.5.5 can be less than the worst case processing delay (plus a safety margin for process variations and simulator errors) by an amount equal to the forward propagation latency of the pipeline control (from *req* \rightarrow ∇sel).

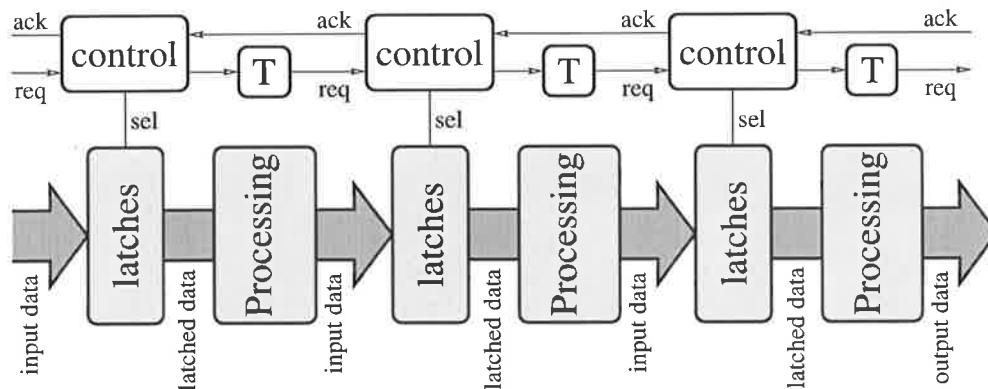


Figure 5.5: A typical delay-modelled pipeline.

The implementation of this delay element is different for 2P circuits than it is for 4P. With a 2P design an approximately equal delay is necessary for both positive and negative transitions, and so a simple inverter chain giving the required delay time can be

used. However for the 4P designs, the delay is only necessary for the positive transition of *reqout*, and the negative transition should ideally have no delay. Obviously an inverter chain *could* still be used, however the delay imposed on the negative propagation would be severely detrimental to the cycle time. Instead, a structure such as that shown in Fig.5.6 can be employed, which enables a positive edge to propagate through the inverter chain, but a negative edge will cause every second inverter to be pulled low, including the output, thereby resulting in a very low propagation latency as desired.

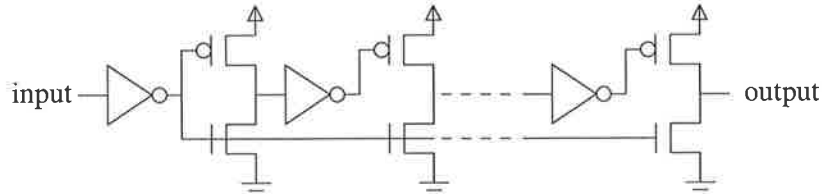


Figure 5.6: A delay element for positive transitions only, as required by 4P controllers.

Note that it is also possible to implement the delay by propagating the output request through a series of gates which mirrors the worst case computation path. By doing this the effects of process variations and simulator errors are reduced, since both control and data delays will vary *almost* identically which cannot be said of the inverter chain delay. This approach is more suited to the use of dynamic logic (discussed in Section 5.3.3.3), since for static logic the worst case computation path cannot always be replicated for both positive and negative transitions of a 2P event, and a 4P control signal cannot usually propagate a fast low signal without altering the delay model. Furthermore, this approach prevents absorbing some of the data modelled delay into the event control (from $req \rightarrow \nabla sel$ in Fig.5.5, which can be used to reduce the stage latency) unless the delay model is truncated, however this would mean that it no longer accurately mirrors the data path which contradicts its intended purpose.

The FIFO circuits of the preceding section (excluding the micropipeline and ECS micropipeline) were simulated using the ES2 technology and are shown in Table 5.2. A peak data processing delay of 10ns was assumed (which includes any safety margins), and an inverter chain delay model was used on each stage's output request.

The latency of these circuits is now almost identical, since their processing delay model has been *absorbed* into the event control from $req \rightarrow \nabla sel$ (which is just greater than the processing time plus latch propagation delay: $11.3ns > 10ns + t_{latch}$). As could also be

Circuit	σ (ns)	λ (ns)	P (mW)	$P\sigma$ (mWns)
Simple 4phase	26.3	11.3	1.6	40.9
Decoupled 4phase	18.0	11.3	2.2	39.6
F-forward micropipeline	19.2	11.3	1.5	28.1
ECS state pipeline	12.5	11.2	1.1	13.8

Table 5.2: Relative performance of four delay modelled pipeline circuits.

expected, the cycle times of these circuits is, for all but the simple 4P (S4P) pipeline, approximately 10ns greater than their FIFO cycle times.

It can be shown that in a pipeline with processing, the cycle time is limited only by the return event processing time (from $\partial ackout \rightarrow \partial ackin$), and *not* by the forward latency (from $\partial reqin \rightarrow \partial reqout$), since this can be incorporated into the delay model. This is why the ECS state pipeline is so fast, since there is a very short return processing time (1.3ns), whereas the decoupled 4P (D4P) pipeline suffers considerably from its RTZ phase (6.7ns).

The advantage of the D4P pipeline over the S4P structure however is now evident. The latter is seen to exhibit a very long cycle time since, as already stated, it can only have each alternate stage processing data, whilst the intermediate stages remain idle. In essence, the processing delays of the current stage *and* the one following are incurred in the cycle time. However, the ECS state pipeline still outperforms the D4P pipeline by a significant margin.

5.3 Precharge pipelines: general concepts

The previous pipeline structures are useful for data computations which incorporate only static logic. However it is often useful to implement dynamic logic structures to reduce power consumption and increase performance. Furthermore, the use of dynamic logic (or at the least, a pipeline with reset and activate phases) enables self-timed computations to be performed. Using a delay-modelled approach in the control path as thus far presented does not allow the speed advantages of self-timing to be utilized, since the worst case delay must be accounted for, and not the typical case which may be significantly faster. For example, a 32 bit self-timed adder has a typical delay for random data of 4.4 adder cells [Gar93], which is 86% faster than the worst case (32 cells), and for an incrementer the improvement is almost 97% (1 incrementer cell versus 32).

It is therefore of importance to devise *precharge pipeline* (PP) structures which can be used together with dynamic logic to implement self-timed (as well as non self-timed) computations.

5.3.1 Dynamic Logic

A general dynamic logic computational block is shown in Fig.5.7. When *act* is low (which implies that no processing is taking place) the *output* signal is precharged high through the *p*-transistor. The computation begins when Δact occurs, prior to which all input signals to the nmos pull-down tree must be valid (coming either directly from the preceding latch stage or through a small amount of static logic). If a pull-down path through the nmos tree exists, then *output* will compute low, otherwise it will remain in the logic high precharge state.

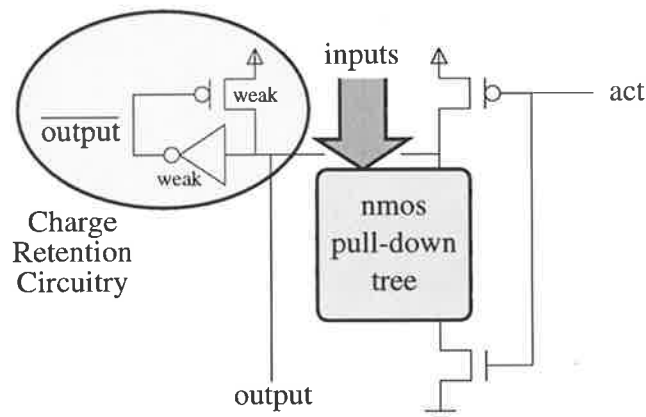


Figure 5.7: A general dynamic logic computational block.

When dynamic logic is used inside a PP structure, it must be remembered that if the present stage has begun its computation with *output* remaining high, but is stalled by the following stage (which may still be computing, or is similarly waiting for its next stage to unstall), it is possible that due to charge leakage *output* will eventually decay low. Although this situation may be extremely rare, if it is at all possible for the pipeline to stall for longer than the charge retention time, then some kind of state holding logic (for the high level) is necessary. The shaded circuitry in the oval of Fig.5.7 provides this function, by maintaining a logic high level through the *weak* pull-up (trickle) transistor via the weak inverter when *output* is high. Note that this will slightly increase the capacitive load on *output* and also the pull-down time.

An alternative is to prevent initiating *act* until the following stage is open [FL96] (so that the results are able to be latched before the influence of charge decay), however this approach is unsuitable for ECS since the increased wait time before activation severely increases the processor cycle time (for the D4P controller the effect is minimal since this can be incorporated into the RTZ phase). Another approach is to compute both *output* and $\overline{\text{output}}$ through an nmos tree, and use each to activate the others trickle transistor. Although useful for some self-timed architectures (in which the pull-down of one of these bits indicates the “completion” of the cell) it has the disadvantage of using approximately twice the area [vBBK⁺95] and requires the inverse of all input signals to be routed.

Dynamic logic has been shown to be of benefit to asynchronous pipelines in terms of both speed and power [McA92, FES94]. By reducing the capacitive load on *output* through removing the static logic pull-up tree, the switching (low) time of the output is improved. Furthermore, due to precharging, the time to a logic high is (obviously) zero. Speed advantages can also result through the use of self-timed logic (see Chapter 6). Power is reduced because of the reduction in capacitance on the *output* node as well as the reduced capacitance on the *input* signals (which only drive 1 transistor per cell instead of 2 for static logic). Furthermore, since the output remains precharged until all inputs are valid, there is no switching power loss due to glitching of the inputs.

At the global pipeline level it can be noted that if static logic is employed, then with all pipeline stages initially open, a signal change at the input may propagate through the entire pipeline, even if no input control event has occurred. This can result in a significant power wastage. One solution is to have each pipeline stage initially closed, however the extra phase of opening *and* closing the latches (instead of just closing them) increases the system latency. Dynamic logic however provides a natural buffer since all outputs are precharged, therefore significant power savings can be made without compromising the system speed.

5.3.2 Requirements of a PP for dynamic logic

Clearly from Fig.5.7 the PP is required to produce an activation (and precharge) signal *act* to the dynamic logic, which starts the computation when high and returns the system to a precharged state when low. This then raises the question: when is it known that the

dynamic logic has precharged or that the computation has completed?

These questions can be answered in two ways depending on the assumptions made on the dynamic computation. If it can be assumed that the precharge time is bounded, then some kind of delay inserted into the PP control can be used to ensure a minimum precharge time, otherwise a signal *pdone* must be returned from the dynamic logic to the PP to indicate when all nodes are precharged (this is necessary for a SI control schema). Similar arguments apply to the computation, with the dynamic logic returning a signal *cdone* to the PP if an unbounded computation time is assumed, or for when the extreme processing variations of a self-timed computation are to be exploited.

Table 5.3 indicates the three different paradigms which result from the above assumptions, dubbed *alpha*, *beta* and *gamma*. The combination of unbounded precharge time and bounded computation time is not included, since it would rarely be of use.

Paradigm	Timing assumptions made	Signals required
Alpha (PP α)	Bounded precharge & computation	act
Beta (PP β)	Bounded precharge & unbounded computation	act, cdone
Gamma (PP γ)	Unbounded precharge & computation	act, cdone, pdone

Table 5.3: Three different design paradigms for precharge pipelines.

PP α is the least robust of the three paradigms and assumes that both the precharge and computation times are bounded. Assuming the former is reasonable since this usually happens in parallel across the dynamic logic array, and assuming the latter (although still reasonable for many applications) prohibits the application of PP α to self-timed structures.

Conversely, PP β is well-suited to implementing self-timed architectures since it requires a completion signal to be generated. The assumption of a bounded precharge time is still reasonable for self-timed architectures, however in the rare instance that it is not then PP γ can be used. This is the most robust paradigm and is necessary if a SI control model is implemented.

The requirements on the PP for generating *act* are obvious: the dynamic nodes cannot be returned to precharge (∇act) until their computed data has been latched into the following stage; and the computation cannot be activated (Δact) until the data nodes have been precharged and new data has arrived (as signalled by the input request of the control schema).

The signal $pdone$ (for $PP\gamma$) is used to govern the activation of Δact on a positive transition only, and the signal $cdone$ (for $PP\gamma,\beta$) is used to govern the output request of the pipeline also on a positive transition. The negative transitions of these signals is therefore unnecessary, and should ideally occur as soon as possible after Δact and ∇act respectively so that they have a minimal effect on the cycle time.

5.3.3 Methods of completion and precharge detection

There are essentially four methods by which $cdone$ and $pdone$ can be generated, each of which gives a different level of robustness in terms of its resilience to process and operating point variations.

5.3.3.1 Self-timed static logic (STSL)

The most robust method of determining when a computation has completed is to generate both the required signal (say, x_i) and its inverse (\bar{x}_i), since under all conditions (barring stuck-at faults) one of these two signals must pull low. By *nand*'ing these two signals together, and then *and*'ing the result for all signals, a self-timed generation of $cdone$ can be produced, as shown in Fig.5.8a. To enable a fast pull-down path from $\nabla act \rightarrow \nabla cdone$, an *or*'ed pull-down transistor in the final *and* gate is used (driven by \overline{act}).

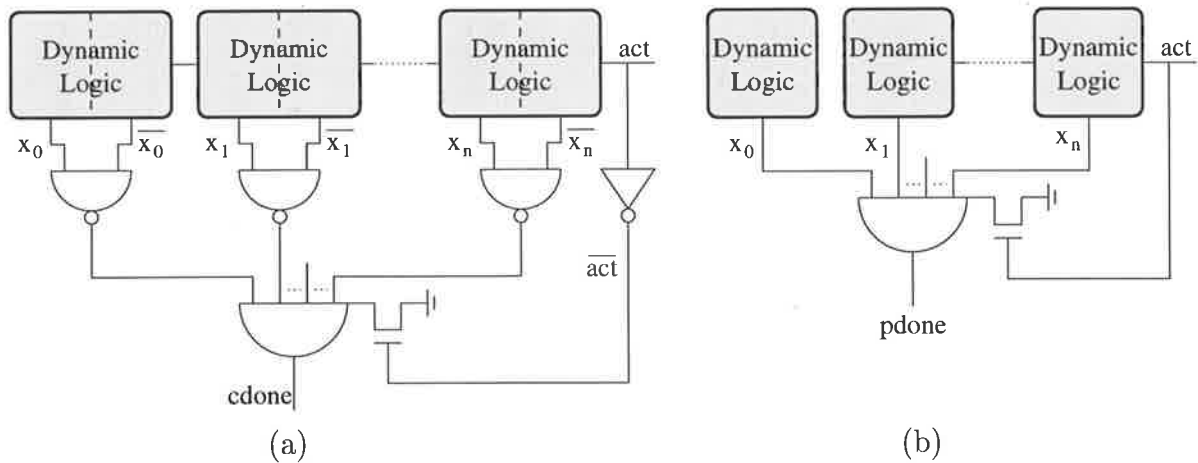


Figure 5.8: A self-timed static logic method for generating (a) $cdone$ and (b) $pdone$.

Figure 5.8b shows a similar method for generating $pdone$. In this instance the precharge is detected by *and*'ing all of the output nodes x_i (and \bar{x}_i), since these must

all be high to indicate completion. As before, an *or*'ed pull-down transistor is used on the *and* gate of *pdone* to produce a very small delay from $\Delta act \rightarrow \nabla pdone$.

Although both of these circuits are suitable for SI control models and self-timed computations, they require a substantial amount of logic in the detection path, especially when one considers that for a large number of bits the high fan-in *and* gate must be re-implemented as a tree of smaller gates. It is therefore evident that self-timed completion detection using static logic is a slow process, and could easily become longer than the dynamic logic computation which would then completely negate its speed advantage over static logic.

5.3.3.2 Self-timed pseudo-nmos logic (STPL)

To overcome the speed deficiency of the self-timed static logic detection, a pseudo-nmos style [WE93, Chapter 5.4.3] implementation can be used as shown in Figure 5.9.

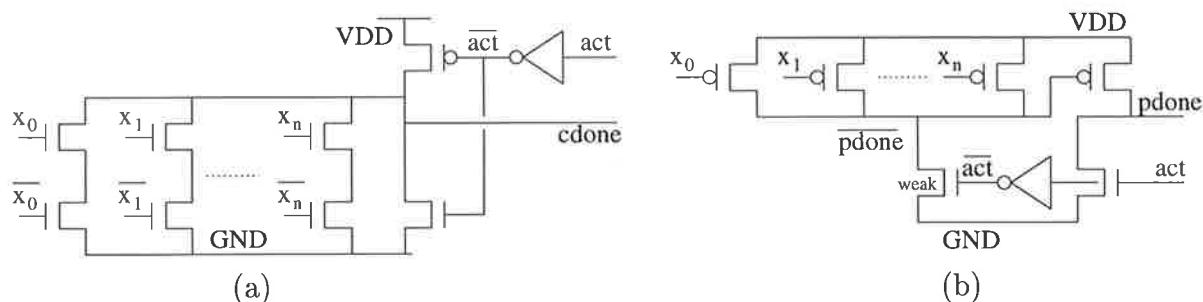


Figure 5.9: A self-timed pseudo-nmos logic method for generating (a) *cdone* and (b) *pdone*.

For generating *cdone*, the initial condition is that all x_i and \bar{x}_i signals are high, and the pull-up transistor is off so that *cdone* is low and no power (other than from leakage currents) is drawn. When *act* goes high, the pull-up transistor turns on and static power is dissipated. This transistor is sized such that in the worst case situation, when only one of the pull-down paths is on, the output *cdone* still provides a suitable logic-low level. When eventually all paths are off, which occurs when for all bits either x_i or \bar{x}_i has computed low, *cdone* will be pulled high. This circuit therefore enables very fast completion detection but at the cost of static power dissipation *only* during the computation phase (which can be minimized by suitable transistor sizing), and is independent of any pipeline stalls. Note also that the n-transistor connected to \bar{act} is used to provide a fast pull-down path during precharge.

The same technique is used for generating $pdone$, however in this instance all nodes are connected to a parallel row of pull-up transistors. If any one of these signals is low, then \overline{pdone} will be pulled high and $pdone$ will stay low (static power is similarly consumed during the precharging process through the weak, appropriately ratio'ed pull-down transistor governed by \overline{act}). As soon as all signals are high, \overline{pdone} will pull low and $pdone$ will rise indicating precharge completion. No static power will then be drawn. Note again that $pdone$ has its pull-down transistor governed by act to provide a fast pull-down time.

5.3.3.3 Computation modelled completion detection (CMCD)

For the case in which one computational node can be identified as exhibiting the worst case pull-down (WCPD) time (as is often the case when each bit computes a similar function) then the detection mechanisms shown in Fig.5.10 can be used to generate $cdone$ and $pdone$. This approach can be expected to closely mirror any circuit deviations arising from variations in process and operating conditions, since the affected pull-down path is replicated in the completion strategy.

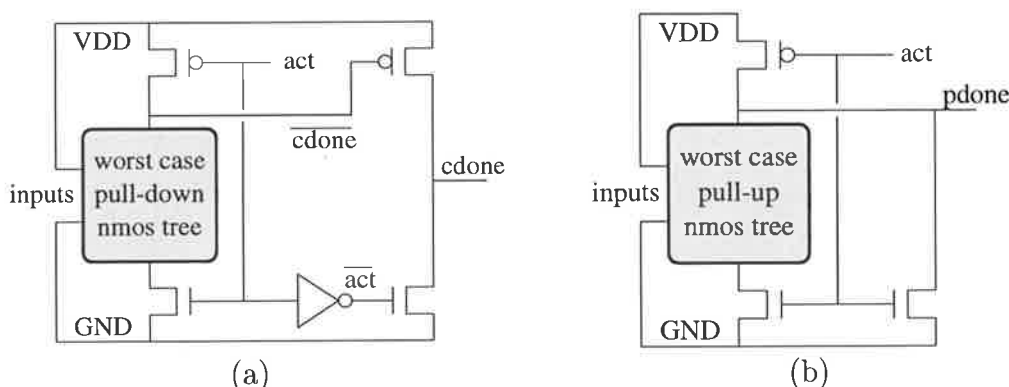


Figure 5.10: A method for generating (a) $cdone$ and (b) $pdone$ which closely models the worst case pull-down time.

In Fig.5.10, act is initially low and hence (since \overline{cdone} is precharged) $cdone$ will be low. Within the WCPD tree, all gate inputs are connected to either GND or VDD, depending on the WCPD path of the computation to be mirrored. When act goes high this path will connect to GND and therefore pull down the node \overline{cdone} , causing $cdone$ to go high (the n-transistor of this inverter is connected to act to enable a rapid pull-down delay from ∇act). Completion detection is therefore rapidly and reliably signalled. Similar arguments apply to the circuit for generating $pdone$, although in this instance the

inverter is not used, and the n-transistor network must be constructed to mirror the worst case pull-up time.

5.3.3.4 Delay modelled completion detection (DMCD)

The simplest and least robust method of signalling completion is to use a delay model, as shown in Fig.5.11. In this instance, a worst case pull-down (and pull-up path for precharging) must be identifiable which, as per the CMCD approach, prevents it from being used for self-timed computations.

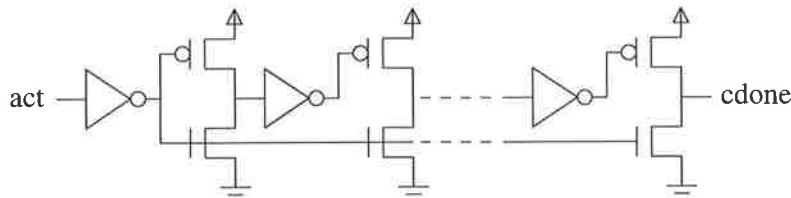


Figure 5.11: A delay modelled method for generating *cdone*.

The signal *cdone* is generated simply by a delay of the *act* signal through the inverter chain, and to enable a fast pull-down path, every second inverter is driven low by \overline{act} (which is connected to its n-transistor). The same circuit can be used for generating *pdone* when preceded by an inverter. Note that this circuit is identical to that of Fig.5.6.

5.3.3.5 Summarizing the completion detection approaches

For generating the *cdone* and *pdone* signals for self-timed computations it is best in terms of speed (which is the primary focus of ECS) to use the STPL mechanism, which is significantly faster than using the static logic approach. Note that the static power dissipation of this circuit can be controlled through transistor sizing, and is only incurred during the *actual* computation time and is therefore not affected by pipeline stalls. The STSL method should not be used for high speed applications unless the data width is small.

If however a worst case node can be identified for the computation and precharge times, then it is faster and more reliable to use the CMCD mechanism. If there is any static logic circuitry present before or after the dynamic logic, then a DMCD approach should be used to model this.

5.4 Decoupled 4P precharge pipelines

The D4P pipeline has been shown to be the fastest of the 4P implementations, and has therefore been used in the construction of α , β , and γ precharge pipelines, with the intention of comparing the performance of these D4P structures against those which can be devised in ECS.

These PP structures can be designed by first considering the more stringent requirements imposed upon the D4P when implementing a $PP\gamma$ and then simplifying this circuit for the other paradigms.

5.4.1 Implementations for $PP\alpha$, $PP\beta$, and $PP\gamma$

Consider firstly the generation of act , for which the conditions for generating its positive and negative transitions can be stated as:

- Δact cannot occur until $\Delta reqout$ of Fig.5.3 has occurred, which indicates that new input data has been latched.
- Δact cannot occur until $\Delta pdone$ has occurred, which indicates that the processing nodes are in a precharged state and are ready for dynamic computation.
- ∇act can be initiated after $\Delta ackout$, which indicates that the data from the previous dynamic computation has been latched. This signal must stay low until $\nabla ackout$ occurs to prevent the subsequent $\Delta pdone$ transition from causing a premature rise in act (resulting from the two preceding requirements).

An asymmetric $cgate$ can be constructed from these three conditions and coupled to the D4P pipeline as shown in Fig.5.12a to create a $PP\gamma$ structure. The output request of this new pipeline is given directly by the 4P $cdone$ signal emerging from the computational block.

A $PP\beta$ structure can be devised simply by removing the $pdone$ signal from the asymmetric $cgate$ (since the precharge time is now assumed bounded), and inserting a delay into the return acknowledge path which enables the precharge time from $\Delta ackout \rightarrow \Delta reqout$ (prior to the $cgate$ for act) to be controlled. Note that if the handshaking control of the RTZ phase is longer than the required precharge time, then no delay element is necessary.

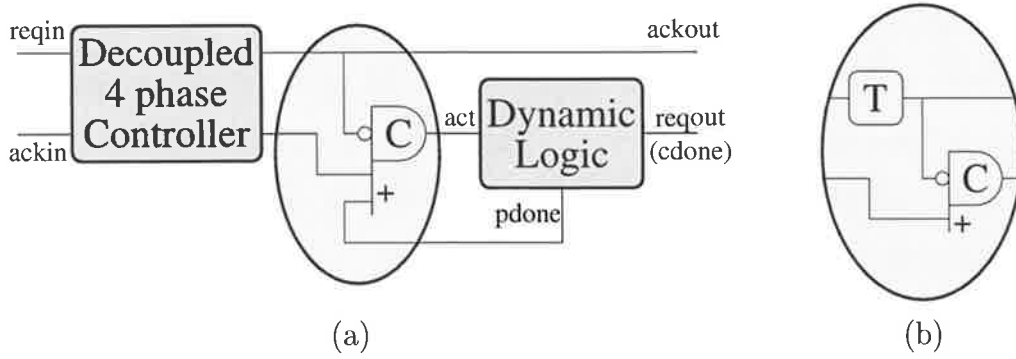


Figure 5.12: (a) $PP\gamma$ and (b) $PP\beta$ structures implemented with a D4P controller.

Otherwise, the additional delay T could be implemented by an inverter chain of delay $T/2$ (since both transitions of $ackout$ occur during the precharge phase), or a positive edge only delay of T . The shaded control section of Fig.5.12a is therefore replaced by that of Fig.5.12b to produce a $PP\beta$ structure.

The simplification into a $PP\alpha$ paradigm is simple. The signal $cdone$ is removed (as a bounded computation time is now assumed), and act is simply delayed (positive edge only) to give the output request signal. Note that the control of the next stage from $\Delta reqin \rightarrow \nabla sel$ can now be incorporated into the overall bounded control delay, hence reducing the pipeline latency and the cycle time.

5.4.2 Performance comparisons

These three PP structures were all simulated in Hspice using the ES2 technology for a 4-stage pipeline with 32 bit buses, and assuming a dynamic computation delay (from $\Delta act \rightarrow$ data valid) of 10ns, and a precharge time (from $\nabla act \rightarrow$ data high) of 3ns. A STPL approach was used for the generation of $cdone$ (resulting in an additional $\approx 1ns$ delay for detection), and a CMCD strategy was used for generating $pdone$ (which incurred a negligible delay since it modelled the 3ns precharge time). Identical drivers for sel and act were also used between each implementation. Table 5.4 provides the results of these simulations, with all values quoted *per stage*.

It can be seen that the latency and cycle times of the $PP\beta$ and $PP\gamma$ structures are almost identical. This is because a D4P controller inherently provides for a precharge time of greater than 3ns (actually, about 6ns) in its RTZ phase, so that no delay element was necessary. As such, the only difference in implementation is in the removal of the “+”

D4P Circuit	σ (ns)	λ (ns)	P (mW)	$P\sigma$ (mWns)
PP α	18.0	11.6	2.6	46.0
PP β	19.4	13.0	2.5	48.7
PP γ	19.6	13.2	2.8	54.5

Table 5.4: Comparison of precharge pipelines (α, β, γ) implemented with a D4P controller.

bar to $pdone$ in the asymmetric $cgate$ (hence the slight difference in speed of 0.2ns). PP α enables a latency closer to the ideal of $t_{latch} + t_{comp}$ (where $t_{comp} = 10ns$ in this example) because the latency of the event control has been absorbed into the delay model. The cycle time is therefore similarly improved by the same amount. Note that in practice a longer delay time would be implemented to provide a suitable safety margin.

5.5 ECS precharge pipelines

The ECS state pipeline discussed in Section 5.1.3 has been shown to give a much faster latency and cycle time than any of the other pipelines. Structures similar to this can be devised which enable all three PP paradigms to be realized. The operation of these pipelines can become complex, therefore it is convenient to begin with the simplest paradigm: PP α .

5.5.1 PP α implementation

The PP α structure in ECS is in many ways similar to that of the D4P approach. In particular, since no $pdone$ or $cdone$ signals are provided, the computation time and precharge times are controlled with a delay element as shown in Fig.5.13. In contrast however to the D4P approach these delay elements now operate on events, and therefore the delay now applies to both transitions rather than just the positive one.

The requirements for generating act are simple: Δact cannot occur until $\partial ackin$ occurs, which signifies that a new operation is underway and the input data is latched; and ∇act cannot occur until $\partial ackout$ occurs, which indicates that the following stage has latched the current data. Note that using these events requires ∇sel to occur before ∇act of the preceding stage has been able to invalidate its data (precharge high). This takes a finite time, so that in practice ∇sel and ∇act can safely occur concurrently. Note that act could simply be generated by the inverse of sel , however by decoupling the activation

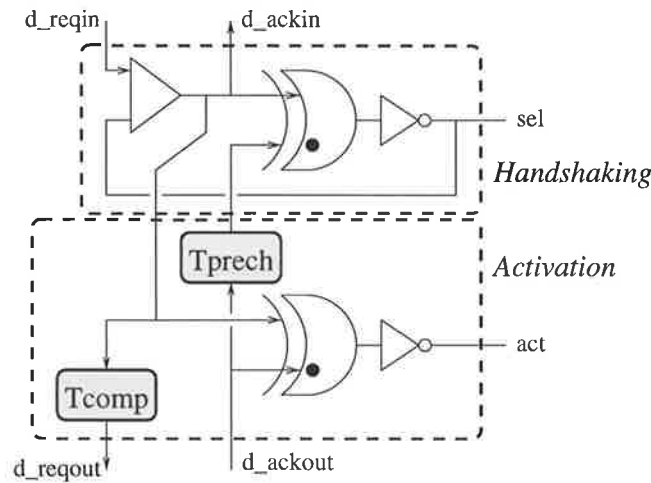


Figure 5.13: An ECS $PP\alpha$ structure.

from the handshaking, an acknowledge can be sent to the previous stage without having to wait for the precharging phase as would have to happen otherwise.

5.5.2 $PP\beta$ implementation

A design of a $PP\beta$ structure is presented in Fig.5.14, and is similarly seen to decouple the handshaking (generation of *sel*) and activation (generation of *act*) phases of the design.

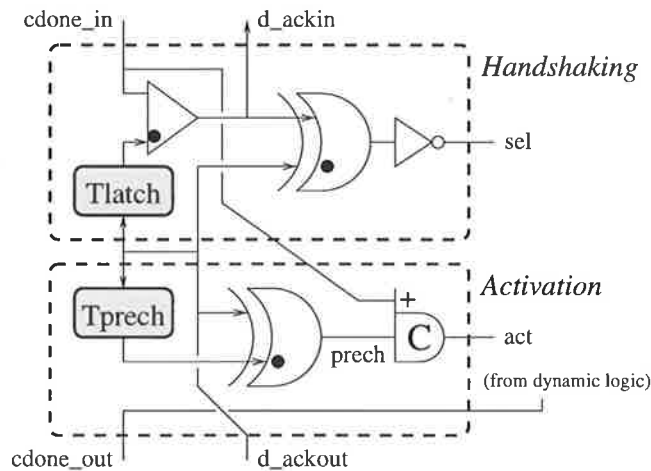


Figure 5.14: An ECS $PP\beta$ structure.

The most striking difference between the handshaking phase of this structure and that of the $PP\alpha$ is that a forward propagating event path is no longer present! This fact is best explained by first considering the $PP\alpha$ structure with a *send* gate placed before *dreqout* governed by *cdone*. This then produces an initial $PP\beta$ structure which can then be modified into this one through the following observations.

A new operation in stage i cannot begin until $\Delta cdone_in$ has occurred, which would then also produce a $\partial reqin$ event to stage i . As such the occurrence of this event is irrelevant, since $\Delta cdone_in$ signifies the same thing (and the $send$ gate is only of use to maintain a forward event flow). However the new operation in stage i still cannot proceed until $\partial ackout$ has occurred. Instead of indicating this occurrence via Δsel as in $PP\alpha$, the event can be temporally *and*'ed with $cdone_in$ using a $send$ gate to begin the new operation. These improvements then result in a faster handshaking operation, since the redundant $\partial ackout \rightarrow \Delta sel$ and $\Delta cdone_in \rightarrow \partial reqin$ delays have been removed. Note that a small delay of $Tlatch$ is necessary to enable a sufficiently long sel pulse to be generated in the worst case (when $\partial ackout$ occurs with $cdone_in$ already high), which can usually be coupled with the precharge delay since this is invariably longer (although they're shown separately in Fig.5.14 for clarity).

In most situations a pipeline will be embedded inside a higher level of abstraction, which may require that an input $\partial reqin$ event be used to trigger the first stage (stage 0), and a $\partial reqout$ event be produced from the last (stage n). In this case, the initial $cdone$ signal to the first stage can be generated by the TE: $cdone_in_0 \leftarrow \partial reqin_0 U \partial ackin_0$; and the final $\partial reqout$ event can be generated by: $\partial reqout_n \leftarrow \overline{\partial ackout_n} \cdot cdone_out_n$ (remembering that the overline indicates a *primed* event, not an inverse).

The activation phase of this circuit is in some ways similar to the $PP\alpha$, which employs an xor gate and a delay to model the precharge time. In this instance however the rise of act is postponed until $\Delta cdone_in$ from the preceding stage has occurred, since this now indicates that new data is available. Furthermore, an asymmetric $agate$ is employed to rapidly produce Δact once $\Delta cdone_in$ has occurred, and the signal from the xor gate is now used to ensure that, regardless of the state of $cdone$, a sufficiently long precharge phase still occurs. Implementing the activation circuitry in this manner enables the latency of the design to be substantially reduced.

5.5.3 $PP\gamma$ implementation

For the $PP\gamma$ implementation shown in Fig.5.15 the same basic handshaking structure as per $PP\beta$ is used, however the activation circuitry deviates from $PP\beta$ because the precharge time is no longer bounded, and hence the $pdone$ signal is used to indicate when

the dynamic logic has precharged. This is incorporated into the asymmetric *cgate* by using $\Delta pdone$ to govern the positive edge of *act*.

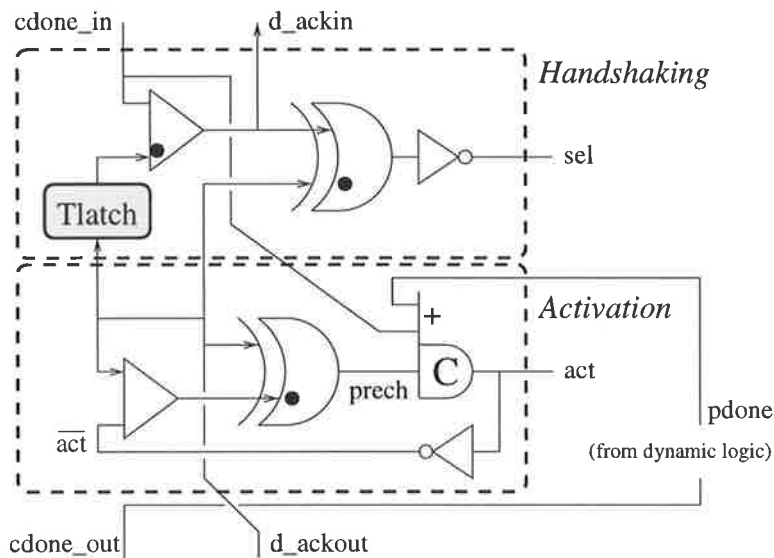


Figure 5.15: An ECS PP γ structure.

The purpose of *prech* now is slightly different from PP β . In this instance it is only necessary for $\nabla prech$ to *initiate* the precharge phase, whereas before it had to *maintain* it, since now $\Delta pdone$ is used to indicate precharge completion ($\Delta prech$ had to be used for this purpose in PP β). It is therefore safe to set *prech* high again as soon as *act* has gone low, and the *send* gate governed by \overline{act} is used to effect this.

5.5.4 Performance comparisons

These three PP structures were all simulated using the ES2 technology with identical design parameters to those given for the D4P pipelines of Section 5.4.2. Table 5.5 provides the results of these simulations, with all values quoted *per stage*.

ECS Circuit	σ (ns)	λ (ns)	P (mW)	$P\sigma$ (mWns)
PP α	14.4	11.4	1.3	19.3
PP β	16.2	11.7	2.7	43.7
PP γ	17.0	11.9	2.7	45.9

Table 5.5: Comparison of precharge pipelines (α, β, γ) implemented using ECS.

As could be expected, the cycle time of PP α is given by the sum of the computation time, latch propagation, an *xor* delay, and the precharge time (with the latency given by

the first two components). The latency is not influenced by the control structure since this has been *absorbed* into the delay model for the computation, and the cycle time is only influenced by the delay of an *xor* gate's negative transition. It is surmised that this ECS $PP\alpha$ implementation is optimal in terms of speed. Interestingly, this circuit also consumes less than half the power of the other paradigms, since there is less control activity and no need for completion detection. Note again that in practice the latency (and cycle time) would be increased by the safety margin incorporated into T_{comp} .

The cycle time of $PP\beta$ is increased by the STPL detection time of the data and the increased delay of the asymmetric *cgate* over the inverter driver used in $PP\alpha$. For $PP\gamma$ this is increased further by the slightly slower *cgate* (which has an additional control signal) and by the increased precharge time which results from having to wait for Δp_{done} before activating Δact (an additional *cgate* delay).

The latencies of $PP\beta$ and $PP\gamma$ are given by the delays of the computation, the completion detection (which occurs in parallel with the data propagation through the latch), and the asymmetric *cgate*. Since the STPL provides for a very fast completion detection which is comparable to the latch propagation delay, these latencies are very similar to those of $PP\alpha$.

5.6 Comparison of ECS and D4P PP structures

By comparing Tables 5.4 and 5.5 it can be seen that the ECS implementations of these PP paradigms give significantly better cycle times and slightly better latencies than the D4P circuits in all cases. Specifically, the cycle times for the α , β , and γ implementations have been improved by 20, 17, and 13 percent respectively for the design parameters used in the simulations.

In fact, the advantages of the ECS implementations become more pronounced as the computation and precharge times decrease. Table 5.6 provides a calculated estimate of the cycle times and latencies of the ECS and D4P circuits when the computation, precharge, and detection times are all reduced to 1ns (if zero delays are assumed, the handshaking circuitry of the ECS designs becomes predominant and limits any further performance increase).

σ (ns)	PP α	PP β	PP γ	λ (ns)	PP α	PP β	PP γ
D4P	9.0	10.2	10.4	D4P	2.6	3.8	4.0
ECS	2.8	4.8	5.6	ECS	2.1	2.5	2.7
%better	69	53	46	%better	19	34	33

Table 5.6: Comparison of ECS and D4P PP structures (α, β, γ) assuming 1ns computation, precharge, and detection delays.

Under these conditions the improved speed performance of the ECS designs are clearly evident, as given by the “%better” row.

5.7 Summary

Numerous pipelining circuits have been presented for use with both static and dynamic logic (or neither in the case of a FIFO), and in all circumstances the 2P ECS implementations have surpassed those of the best 4P designs (as well as any other 2P pipelines previously reported). In particular, improvements of up to 64% in the case of static logic and 69% for dynamic logic have been demonstrated, which translates into approximately a three times speed up. Furthermore, the power consumption of these ECS circuits is also considerably less than the D4P implementations.

Some fast techniques for implementing the completion detection necessary for dynamic logic have also been presented. In particular, the STPL approach allows for self-timed architectures to be implemented without incurring the excessive overhead resulting from conventional static logic detection mechanisms.

Chapter 6

Self-Timed Architectures

THE previous chapter on asynchronous pipelines made numerous references to the use of self-timed architectures, and in particular to their ability to execute at a data dependent rate. These architectures can therefore take advantage of the best and typical case computation times (being data dependent), which for certain applications can be significantly faster than for the worst case situation. Furthermore, if the data is such that the worst case occurrence is infrequent, then the average case computation time will approach the best case. In contrast, a BD architecture will still be governed by the worst case computation time regardless of the data, so that *typically* a much slower execution rate results.

As an illustration of a data dependent computation, consider the conditions for generating a full adder cell's output carry from two input signals and an input carry, as detailed in Table 6.1.

a	b	C_{out}
0	0	0
0	1	C_{in}
1	0	C_{in}
1	1	1

Table 6.1: Conditions for generating the output carry of a full adder.

When a is equal to b there is no need for C_{in} to propagate through to the next bit, since C_{out} can be computed directly from the inputs. In fact, if this is the case for all of the bits in the input operands, then there is *no* need for any carry propagation to occur at all!

A self-timed architecture can take advantage of this scenario by generating a *validity* signal for all bits which are free of carry propagation. Any carries (and corresponding validity bits) which *are* generated only need to propagate as far as the next cell in which a carry (and validity) was also generated. It can be shown that for random input data the average carry propagation length of an adder is approximately $0.9 \log_2 n$ for input operands with n bits [Gar93], which is considerably faster than the worst case BD propagation of n bits, especially when n is large (such as in cryptography applications). One drawback of self-timing is the need for completion detection across all bits of the computation (from each bit's validity signal), however the techniques of the previous chapter, and in particular the STPL mechanism, enable this function to be performed rapidly.

Although synchronous systems (and asynchronous systems employing a BD computational model) require the full n bit propagation to be accommodated, techniques such as carry look-ahead [WS58], carry selection [Bed62], and parallel decomposition [BK82] can be used to decrease the propagation delay of the carry chain, though often at the cost of an increase in area and a loss of regularity. Despite these techniques, self-timed architectures still enable a performance benefit for the best and typical cases especially for operands with large n . Only when the typical and worst case scenarios are similar does the self-timed approach become inconsequential.

This chapter will present the design and implementation of a range of self-timed subsystems which are commonly used in a number of processor architectures, with particular emphasis on those structures used in critical portions of microprocessors (such as PC incrementing, branch target calculations, and integer processing units). Furthermore a faster and lower area method of self-timing (dubbed pseudo self-timing) is also presented.

6.1 Strict self-timing requirements

For an operation to be self-timed, it requires there to be at least three states for any one signal, as indicated in Table 6.2. Two states are used to convey the conventional binary logic information (states 0 and 1), and the third state is used to encode the timing, or data validity of the signal (state 2). It is therefore evident that to design a strictly self-timed system, two wires are need for each bit to encode these three states. This convention is known as *dual rail* (DR) logic, and although circuits based on this paradigm are highly

robust (since all signals have timing encoded with them) they have also been shown to require approximately twice the area of their single rail (SR) counterparts [vBBK⁺95].

State	Meaning
0	Logic 0
1	Logic 1
2	Invalid
(3)	(<i>unused</i>)

Table 6.2: Three states required for implementing self-timed logic.

Note that since two wires are used to encode the three states, there is a fourth *unused* state which can be used for other purposes. In particular, an *error* state may be encoded here which indicates when both a logic high *and* a logic low level are erroneously being transmitted on a signal (as in Table 6.3). Alternatively, this extra state may be used to implement ternary logic, in which a bit may take the values of 0, 1, or 2. This technique reduces the bit length of words by $\log_2 3 \approx 1.6$ times, however since most self-timed computations are proportional to the log of the bit size (as in the adder), this then results in a *constant* reduction in propagation length of only $\log_2 1.6 \approx 0.7$ bits regardless of the data width. Given that the implementation of VLSI gates in ternary logic is more complex (and slower) than for binary logic, the net result may well be an overall *increase* in the propagation delay.

Dual rail logic must be used to implement strictly self-timed systems, however as stated in Section 2.4 the ECS design paradigm uses single rail data. How then is it possible to implement self-timed computations in such an environment?

One solution is to compromise the requirements of self-timed systems as follows. The input and output signals of the computation are provided in SR format together with an activation signal *act* which indicates their validity, and a completion signal *cdone* is generated when the output data is valid. However *within* the computation itself any propagating signals between bit cells are implemented in DR format, so that the completion of the computation can be detected. This approach then enables an *internally* self-timed unit to interface to the SR paradigm of ECS, and in particular, may be utilized directly as a computational element in the pipeline structures of Chapter 5.

6.2 Designing and utilizing self-timed units



Dynamic logic is ideal for implementing self-timed logic, since during the precharge phase all DR nodes are forced (in parallel) into the *invalid* state as required. When the computation is activated, the logic is designed such that only one of the two wires encoding the data can change state (the precharged dynamic node pulls low), indicating that a *valid* logic level is now present. This will then initiate a change in state of the following cell's DR signal if one hasn't already occurred independently of this. Although this functionality can be implemented with static logic, it becomes more cumbersome and is significantly slower than the dynamic logic approach.

The environment supplies its data in SR form and initiates Δact when these are valid, which then causes the self-timed computation to begin by discharging the relevant dynamic nodes. Once complete, the self-timed unit will generate a *cdone* signal back to the environment using the STPL approach of Section 5.3.3.2, and some time later (depending on the environment's structure), ∇act will occur to invalidate the DR data through precharging in preparation for the next operation.

In a pipelined architecture self-timed logic is of no use in improving throughput, but it can be beneficial in improving the latency. This latter fact is obvious, since by reducing the typical propagation length the computation time is decreased, however to understand why the throughput is not improved it is worth considering again the operation of a 32 bit adder.

In a synchronous system implementing two stages each of 16 bits will approximately double the throughput, since the worst case propagation delay of each stage is halved. However, a self-timed system is governed by the typical case, which gives a reduction from 4.5 bits to $0.9 \log_2 16 \approx 3.6$ bits. This is only a marginal improvement in throughput, and has occurred at the expense of a significant increase in the latency to $2 * 3.6 = 7.2$ bits *plus* the pipeline control delay (totalling over 60% more for just one extra stage)!

In the extreme case a non self-timed system can be pipelined down to a small number of bits per stage which can all be computed in parallel, therefore resulting in a very high throughput, whereas the self-timed approach cannot match this even when pipelined down to 1 bit per stage, since there is an overhead in completion detection. Furthermore, such an implementation will also exhibit a significantly longer latency than the non self-timed

design (which is evident by extrapolating the two stage example above).

Therefore a self-timed circuit is of little use when the throughput of the design is of prime concern, since this can be better achieved using conventional SR logic techniques. The advantage of self-timing derives from its latency improvement, and should be limited in use to situations in which such an improvement is beneficial even if it results in a reduction in throughput.

6.3 Adder Structures

A very commonly used structure which can be self-timed is that of an adder, whose carry propagation requirements were given in Table 6.1. Using this table and the state allocation requirements of Table 6.2, it is possible to devise an encoding for the DR carry signals as shown in Table 6.3.

Wires		State encoded
c0out	c1out	
1	1	not valid
1	0	logic 1
0	1	logic 0
0	0	error

Table 6.3: State encoding of dual rail carry propagation signals.

Encoding the invalid state as “11” enables the precharge phase to invalidate the carry signals directly. A “0” or “1” propagation of the carry can then be detected (indicating validity) by a logic low level on the respective wire *c0out* or *c1out*. The resulting state table for generating an output carry from the two input carry wires and the input operands *a* and *b* is shown in Table 6.4.

If *c0in* and *c1in* are both low then an error has occurred, and the value of *cout* is irrelevant. If *a* is equal to *b* then a carry of either zero or one is generated onto the appropriate wire, otherwise the input carries (*cin*) are propagated directly to the output carries (*cout*). These signals can be determined from the state table as follows:

$$\begin{aligned}
 c0out &= g + c0in.\bar{h} = g + c0in.p \\
 c1out &= h + c1in.\bar{g} = h + c1in.p
 \end{aligned}
 \tag{6.1}$$

c0in	c1in	a	b	c0out	c1out
0	0	0	0	-	-
0	0	0	1	-	-
0	0	1	0	-	-
0	0	1	1	-	-
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	0

Table 6.4: State table for the dual rail carry propagation signals.

with all carry signals initially precharged high and $g = a.b$, $h = \bar{a}.\bar{b}$, and $p = a \oplus b = \overline{g+h}$. The sum value, which for SR data is given by $s = p \oplus cin$, can now be generated using DR carry signals as:

$$s = p \oplus c0in = p \oplus \overline{c1in}$$

It is also necessary to detect when each bit has completed its carry propagation. Since this occurs when either of the DR signals goes low, this can be generated simply by *nand*'ing $c0out$ and $c1out$.

6.3.1 Self-timed ripple carry (RC) implementation

In implementing an adder cell based on the DR carry ripple equations it is imperative to minimize the carry propagation time across the cell. An implementation which results in even a small improvement in this delay is of relevance, since the effect is magnified n times when extrapolated across the full operand width (in the worst case), and $0.9 \log_2 n$ times for the typical case. A short propagation time can be achieved by providing a fast pull-down path for the dynamic logic with a minimal capacitive loading. Furthermore, since any subsequently driven gates only require a fast pull-up time, they can be implemented with a large p:n width ratio.

The carry equations could be implemented simply by generating c_{out} with a complex dynamic logic gate, however this approach results in a significant load on the carry signal and a 3 transistor pull-down path (including the activate transistor beneath the nmos tree). Instead, the implementation shown in Fig.6.1 is used.

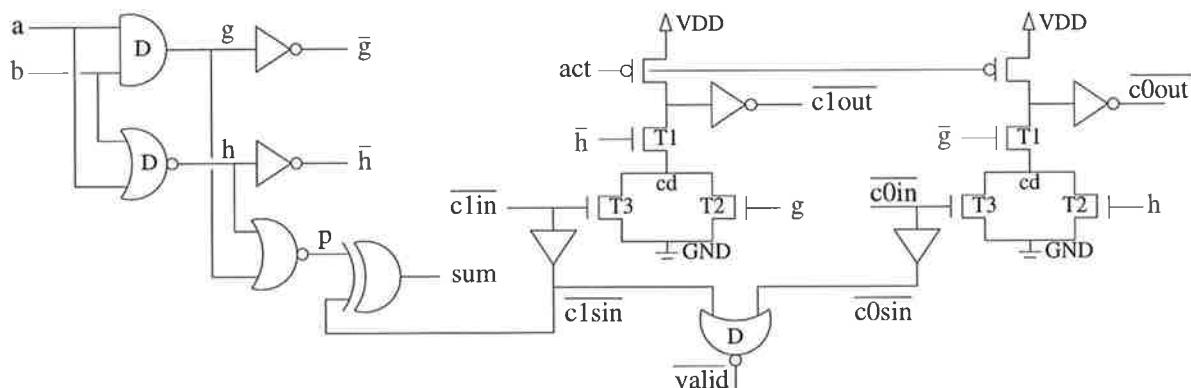


Figure 6.1: A self-timed ripple carry implementation of an adder cell.

In essence, the combinational terms g , h , and their inverses are computed using dynamic logic (as indicated by the D symbol inside the gates), which are then applied to the dynamic computation of c_{out} . Note also that since p is computed from the dynamic nodes of g and h , any spurious transitions on the inputs a and b will be prevented from propagating through to the output sum , thereby conveniently reducing power consumption.

The implementation shown for c_{out} varies from conventional dynamic circuits in that there is no activation transistor governed by act (as per Fig.5.7). Instead, all signals to the dynamic logic block are initially low which allows this transistor to be removed and hence reduces the carry propagation delay. Note however that the signal to transistor T2 is in fact initially *high*, but since T1 is off the precharging phase can still reliably occur.

Once Δact occurs, T2 turns off (depending on the values of a and b) *before* T1 can turn on, so that an erroneous *glitching* pull-down path is prevented. If however a pull-down path is validly activated (either through T2 staying on or when the input signal \overline{cin} goes high), then the relevant c_{out} signal will go low through only 2 transistors. By sizing the inverter to $\overline{c_{out}}$ with a large $p:n$ ratio a very fast carry propagation delay results.

Although this design provides a short propagation delay there is a potential problem with regards to charge distribution. During activation, if T1 turns on but T2 and T3 remain off, then the charge stored on c_{out} will redistribute onto node cd , and reduce the logic high level of c_{out} to:

$$V_{cout} = V_{dd} \frac{C_{cout}}{C_{cout} + C_{cd}}$$

where C_x represents the capacitance of node x . To ensure a suitable logic high level on the output the capacitance of node cd must be small compared to $cout$ (4 times smaller for a 20% reduction in V_{cout}). The VLSI implementation of this carry generation circuit is therefore a critical design issue.

Note also that if T1 is moved to the bottom of the dynamic logic tree then the charge distribution problem is avoided (T1 then acts as an activation transistor). Since this results in a slightly slower carry propagation delay, and since the charge sharing problem can be controlled, transistor T1 is left in the design at the top of the nmos stack.

6.3.2 Self-timed ripple select (RS) implementation

One technique used in the design of single rail adder structures is to precompute the output carry for both possibilities of the input carry (low or high), and then *select* the appropriate one through a multiplexer when the input carry value arrives. A similar ideal can be used for implementing a ST adder, by precomputing the carry values at each bit (as opposed to a group of bits in the SR carry select approach) and selecting the appropriate one when the input carry becomes valid. For such a structure, the ripple carry equations of Eq.6.1 are re-structured as:

$$\begin{aligned} c0out &= c0in.\bar{h} + \overline{c0in.g} \\ c1out &= c1in.\bar{g} + \overline{c1in.h} \end{aligned}$$

The input carry signal can now be used to select between one of two precomputed outputs, as shown in Fig.6.2. Dynamic logic is again used for computing the multiplexer inputs to ensure that all $cout$ signals are initially high (invalid) after precharge. Note that the signals p , sum , and \overline{valid} are all generated as shown in Fig.6.1.

This design may be enhanced further by removing the inverter delay from cin to \overline{cin} . This is achieved by implementing another layer of multiplexing which produces \overline{cin} directly from the inverse of the inputs to the multiplexers of Fig.6.2. This implementation is referred to as a dual ripple select (DRS) adder.

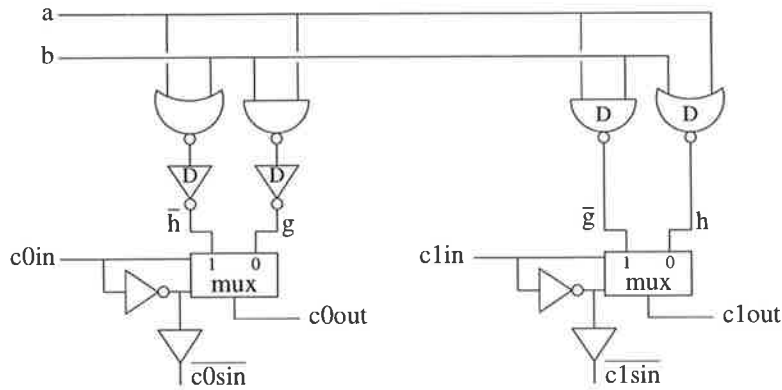


Figure 6.2: A self-timed ripple select implementation of an adder cell.

6.3.3 Comparison of ST adders

These three self-timed 32 bit adder structures have been implemented with STPL completion detection using the ES2 technology, and simulated in Hspice (level 13) using similarly sized transistors for the dynamic pull-down nodes (approximately a 9:1 n-transistor width:length ratio) and pull-up inverters (using an 11:1 p-transistor width:length ratio, with a small n-transistor). The results are shown in Table 6.5.

ST Adder Design	Delay (ns)			Area/cell (transistors)
	best	ave	worst	
RC	2.5	3.5	10.6	50
RS	3.2	6.5	27.5	56
DRS	3.2	5.0	17.9	64

Table 6.5: Comparison of three self-timed adders.

The best and worst case scenarios are for carry propagation across 0 and 32 bits respectively, and the average case assumes a carry propagation length of 5 bits (which is close to the random data average of 4.5 bits). All times quoted include the delay of the STPL completion detection, which is approximately 1ns.

It is clearly evident that the RC design gives by far the best speed performance, being almost 3 times faster than RS and 2 times faster than DRS, as well as utilizing fewer transistors (implying less power and area usage). The reason for this is that the multiplexer outputs of the RS approach drive a higher load (up to 4 transistors versus 2 transistors for the RC design) and with less drive strength than the dynamic computation of the RC approach.

Nonetheless, the speed improvement of the DRS over the RS approach is evident, being 29% faster. This is due to the removal of an inverter delay per stage and the smaller load on the multiplexer (since the transistors of the extra transmission gate are smaller than is otherwise needed for the inverter).

6.3.4 Pseudo self-timing (PST)

As stated in Section 6.1, a strict self-timing environment requires all signals to use dual rail encoding, whereas the adder designs thus far presented assume single rail input signals and initiate the *internally* self-timed DR carry propagation after Δ_{act} . This process can be taken one step further by using SR signals for the carry propagation, and initiating a *matched path* strategy for completion detection once Δ_{act} occurs. Figure 6.3 illustrates this approach.

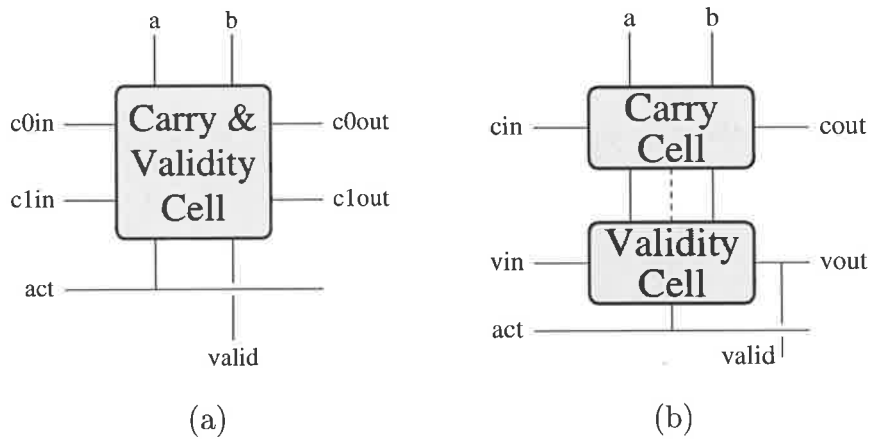


Figure 6.3: (a) A self-timed and (b) a pseudo self-timed generalized view of an adder cell.

A PST design decouples the carry propagation circuitry from the validity detection, whereas a ST design unifies these two functions. This decoupling enables conventional SR designs to be used for the addition process resulting in a reduced load on the *cout* signals (since they no longer initiate cell validity), which ought to subsequently improve the propagation time and reduce the area usage. Furthermore, the load on the *act* signal can be significantly reduced since it must only drive the validity detection, although it may still be used for the carry generation if dynamic logic is preferred. If however static logic is used in their computation then the sum values will emerge from the adder before Δ_{cdone} by approximately the same margin as the inputs arrive before Δ_{act} . This then

ensures greater reliability in their validity and more flexibility in when the environment can process the outputs.

The disadvantage of the PST approach is that if static logic is used for the carry generation then the validity circuitry, which only propagates a high level, must be implemented to give at least the same stage delay as the carry circuitry for both the high *and* low logic levels, which significantly restricts its implementation. Another problem is that there is less resilience to operating variations since the validity is no longer directly computed from the carry propagation, although having these beside each other in the layout, as would be expected, renders this problem inconsequential.

To illustrate the PST methodology, it is worth re-considering the ST adder designs thus far presented.

6.3.5 PST ripple carry (RC) implementation

The SR carry equation $c_{out} = g + p.c_{in}$ can now be implemented directly using the same dynamic logic block as in the ST design, and is shown in Fig.6.4. Note that in this instance both \bar{g} and \bar{p} are low after precharging, which prevents the potential glitching path to ground that was present in the ST design.

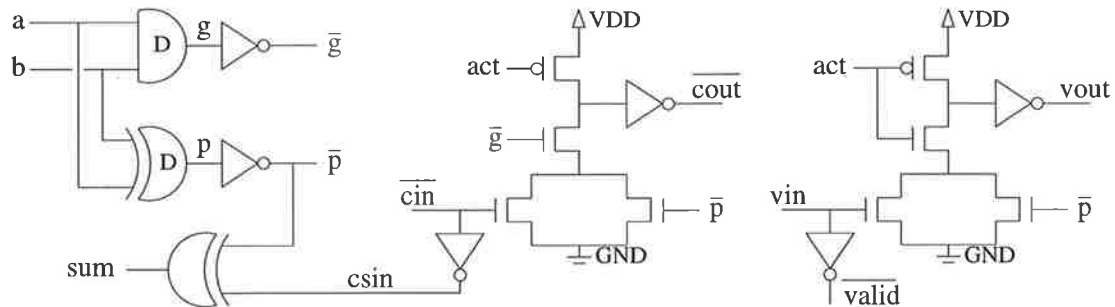


Figure 6.4: A pseudo self-timed ripple carry implementation of an adder cell.

The matched path validity detection must now be designed to correlate to the carry propagation path. Since the carry value of a cell is known when $p = 0$, an input validity signal (which indicates that the previous carry value is valid) only needs to propagate to the output validity signal when $p = 1$, so that: $v_{out} = \bar{p} + vin$. However, since all v_{out} signals must pull low when $act = 1$, v_{out} must actually be computed as: $v_{out} = (\bar{p} + vin).act$. This equation is in the same form as that for generating $\overline{c_{out}}$, and can therefore be implemented using the same logic structure so that both propagation paths

are *matched*. The inverters to \overline{cin} (which buffers the sum generation from the critical carry path) and \overline{valid} are identical, and are designed to provide a low load to \overline{cin} and \overline{vin} respectively.

6.3.6 PST ripple select (RS) implementation

To implement a ripple select adder the carry equation must first be re-structured to give precomputed terms for \overline{cin} and \overline{cin} :

$$cout = cin.\bar{h} + \overline{cin}.g$$

As opposed to the ST implementation of a RS adder, there is now no need for the multiplexer inputs for \overline{cout} to be dynamic. This then reduces the load on act which can be especially beneficial for large data widths. The validity equation must also be re-structured to give the appropriate multiplexer inputs, which must again be forced low when $act = 0$.

$$vout = vin.act + \overline{vin}.(act.\bar{p})$$

Figure 6.5 shows an implementation of the PST RS adder in which the inverse of the carry and validity signals are propagated.

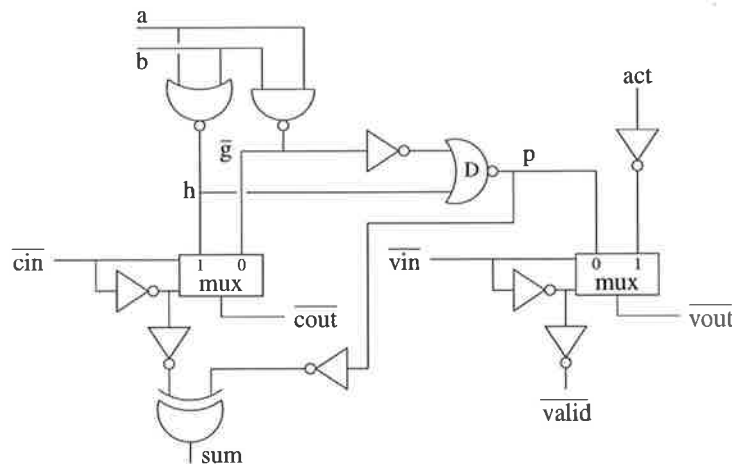


Figure 6.5: A pseudo self-timed ripple select implementation of an adder cell.

The carry signals now compute independently of act since static logic is employed. This has the benefit of producing the output sum earlier than the output validity and reducing the load on act , but suffers in that spurious transitions on a and b now propagate

through the design and waste power. If this is a problem then dynamic logic could again be used for generating \bar{g} and h (which then also requires a dynamic *xor* gate to generate p from a and b).

A dual ripple select (DRS) adder can again be formed with another layer of multiplexers for *cout* and *vout* with inverted input signals.

6.3.7 Comparison of PST and ST adders

These three PST adder structures have been simulated under the same conditions as those of Section 6.3.3, and the results are shown in Table 6.6.

PST Adder Design	Delay (ns)			Area/cell (transistors)
	best	ave	worst	
RC	1.9	2.8	9.9	40
RS	3.0	6.2	26.0	38
DRS	2.7	4.4	16.8	50

Table 6.6: Comparison of three pseudo self-timed adders.

As expected, the *relative* performance of these three adders is identical to the ST designs of Table 6.5, with the RC approach giving approximately 2 and 3 times faster latency than the DRS and RS approaches respectively.

The PST RC design exhibits the same propagation delay as the ST design which is expected since they both use the same circuitry. If however the carry signals to the sum and completion detection of both designs weren't buffered, then the PST circuit would have a faster propagation time since there is less load on the carry signal (no validity circuit to trigger). The PST design is however still slightly faster, due to the removal of the $cin \rightarrow vout$ path (a constant offset). Furthermore, for a slight increase in speed the PST design also uses 20% fewer transistors, which implies a reduction in both power consumption and area usage.

The RS and DRS designs using PST are also slightly faster than their ST counterparts, due again to the removal of the $cin \rightarrow vout$ path. For these designs the area saving using PST is even greater, being 32% and 22% respectively. It can be concluded then that PST adder circuits are more favourable than ST circuits for high speed applications, and probably for low power and area applications as well.

As a further comparison the ST adder design used in the AMULET processor at

Wires		State encoded
c0out	c1out	
0	0	not valid
0	1	logic 1
1	0	logic 0
1	1	error

Table 6.7: State encoding of the AMULET carry propagation signals.

Manchester University was implemented [Gar93, FDG⁺93]. This enables the relative performance of the fastest ST and PST adders (being the RC versions) thus presented to be determined, as well as the performance of the STPL detection mechanism (since a different scheme is used in their design). The AMULET adder uses the carry encoding scheme of Table 6.7 (which is the inverse of that given in Table 6.3), and the circuit implementation for addition is reproduced in Fig.6.6.

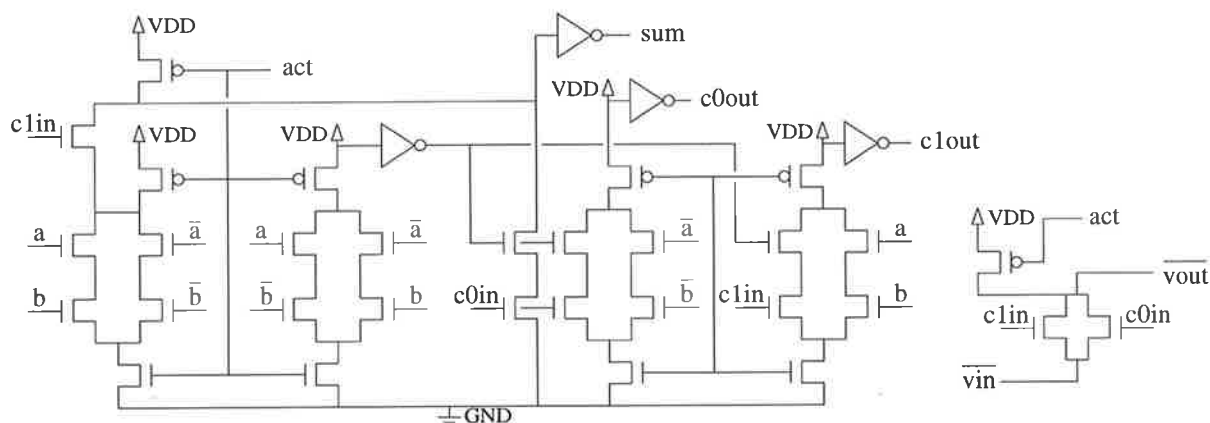


Figure 6.6: The adder cell and validity detection used in the AMULET processor.

This design uses a significant amount of precharging in its dynamic implementation, resulting in a large load on the *act* signal but providing a fast propagation time for the carries. The dynamic pull-down path for the inverse carry signal requires 3 transistors as opposed to only 2 in the ECS designs, which indicates that its speed will be slower.

The completion strategy implemented for the AMULET processor is indicated by the right-most pass transistor structure. To prevent excessive RC delays across the 32 bit propagation path, four of these gates are connected serially from every fourth bit, and the resulting eight outputs are logically *nor*'ed (through 3 *nor* and 1 dynamic *and* gate) to produce the *cdone* signal. This approach then prevents the best case scenarios from being properly utilized, since detection at the first stage of the chain must still propagate through 3 other pass transistors and a tree of gates before $\Delta cdone$ occurs.

The AMULET adder was implemented using identical design conditions and parameters to those of the previous sections, and the simulation results for both the AMULET detection mechanism and STPL are shown in Table 6.8. For reference the fastest ECS design and one of the fastest known adders are also reproduced.

Adder Design	Delay (ns)			Area/cell (transistors)
	best	ave	worst	
AMULET (n-pass)	6.0	6.3	16.5	43
AMULET (STPL)	2.2	3.7	15.4	44
ECS (PST)	1.9	2.8	9.9	40
MODL	3.1 simulated			$0.51mm^2$

Table 6.8: Comparison of ECS and AMULET adder designs (as well as their detection mechanisms) and a known fast adder scaled to a $0.7\mu m$ technology.

The STPL validity detection has resulted in a distinct speed advantage for the AMULET adder, giving 63%, 41%, and 7% improvements for the best, typical, and worst case scenarios respectively. However the PST adder is still significantly faster, giving improvements of 68%, 56%, and 40% respectively over the AMULET adder. It is also marginally smaller.

The MODL design (multiple-output domino logic) of [HF89] was implemented in a $0.9\mu m$ technology, and their simulation results as quoted have been appropriately scaled to indicate performance in a $0.7\mu m$ technology. The delay has been scaled by α^2 and the area has been scaled by α (where $\alpha = 0.7/0.9$) according to the lateral scaling method applicable to sub-micron devices [WE93, Section 4.13]. In the typical case the PST adder is almost 10% faster, and given that its VLSI layout (including STPL control) occupies only $0.12mm^2$, it is also 76% smaller.

6.4 Incrementer structures

A 32 bit self-timed adder gains a performance advantage over a single rail adder because the average carry propagation length is approximately 7 times less than for the worst case, which must always be accommodated for by the latter. With regards to an incrementer, which simply adds “1” to an input operand a , this advantage is even greater. The worst case propagation length for an incrementer is still 32 bits, but the average case is given by $\sum_{i=1}^{32} (i-1)/2^i \approx 1$ bit: a 32 times improvement! Furthermore, although the average

propagation length is 1 bit, the median case (most often occurring) requires no carry propagation at all (when the LSB of a is zero), which occurs 50% of the time for random data.

An incrementer is essentially nothing more than an adder, but with the operand b masked to zero and an input carry of one. In a single rail paradigm the only disadvantage in using an adder as an incrementer is a slight increase in propagation delay. This is because a dedicated SR incrementer can take advantage of the known state of $b = 0$, and hence reduce the carry logic and improve the propagation time across the full data width.

However, a self-timed adder suffers yet *another* problem when used as an incrementer. Consider for example the case in which $a = 11 \dots 10$ is to be incremented. Since $b = 0$, a carry of zero (∇c_{out}) will have to propagate through all of the upper 31 bits before completion is detected! However it is obvious that since the LSB is zero, no propagation should be necessary. The ST adder can therefore substantially increase the average propagation delay when used as an incrementer, since carries must propagate through all of the “1” chains present in a . This same argument applies to a PST adder, although in this instance the propagation is for v_{out} , and not necessarily for c_{out} (depending on whether static or dynamic logic is used for the latter).

It is therefore worth implementing a ST incrementer which does not suffer from unnecessary carry propagation delays through any “1” chains present after the first zero.

6.4.1 Self-timed incrementer

Given that $b = 0$, the carry and sum equation for the incrementer cell can be reduced to:

$$\begin{aligned} c_{out} &= a \cdot cin \\ sum &= a \oplus cin \end{aligned}$$

A ST incrementer should not have to propagate a carry “1” any further than the first zero, since it is known that after this bit all subsequent carries will be low. This can be effected by first setting cin low for all bits when $act = 0$ (implying the use of dynamic logic, and effectively *and*’ing the above equation for c_{out} with act), since this prevents any unnecessary carry “0” propagations and also provides a correct initial cin value to the sum . Furthermore, it is no longer necessary to implement DR logic for the carries, since the only possible propagation is now for $c_{out} = 1$.

It is also necessary to determine how completion can be detected given that SR carry propagation is now used. Since it is known that for all incrementer cells $cin = 0$ initially, the validity can be detected by the first cell which has $cin = 1$ (implying that a carry has propagated to this cell) and $a = 0$ (implying that the carry propagation chain is finished), so that: $valid = \bar{a}.cin$. This cell is unique, and as such a simple dynamic NOR gate can be used to generate \overline{cdone} from the 32 $valid$ signals. This approach draws no static current during operation, and therefore consumes less power than the STPL detection mechanism which has to be employed if an adder were used as an incrementer.

Figure 6.7 shows one possible implementation of the ST incrementer. As per the fastest of the ST adder designs, the inputs to the nmos tree for \overline{cout} are forced low initially, thereby removing the need for an activation transistor. Similar arguments with respect to charge distribution between nodes cd and \overline{cout} are therefore also of relevance here. Since it is known that for an incrementer $cin = 1$ for the first cell, the logic for that unit can be reduced, as shown by the left-most circuitry for $cout_0$, $valid_0$, and sum_0 .

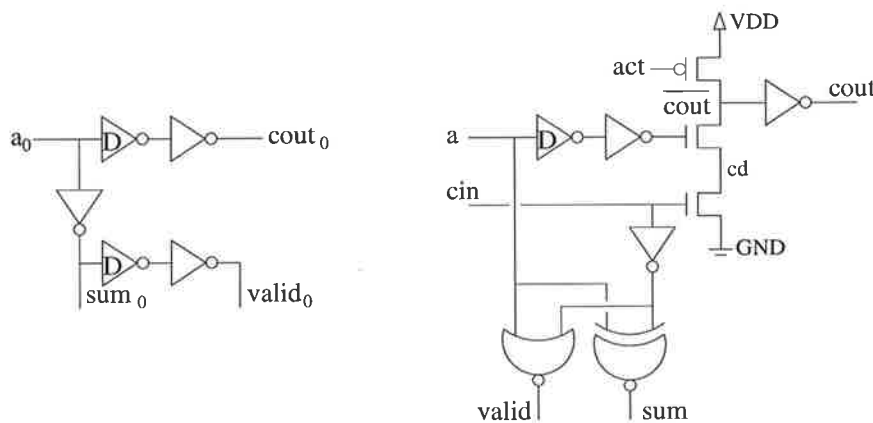


Figure 6.7: A self-timed incrementer without unnecessary carry propagations.

6.4.2 Incrementer performance

The incrementer has been implemented in the ES2 technology with the same design parameters used for the adders. The Hspice (level 13) simulation results are shown in Table 6.9 for the best, average, and worst case scenarios.

Incrementer Design	Delay (ns)			Area/cell (transistors)
	best	average	worst	
Self-timed	1.1	2.2	10.6	22 (12)

Table 6.9: Simulation results of a self-timed incrementer.

If a PST adder were used instead, the average delay time would be over approximately 5 cells (the typical maximum length of a carry “1” chain), rather than for just 1 cell in the incrementer. As such, by comparing these results with those of Table 6.6, it is evident that the best and average case delays for the special-purpose incrementer are better than those of a PST adder. The worst case scenario is slightly slower, but would only happen 1 in 2^{32} times. In the most frequent case of no propagation, the special-purpose incrementer is 42% faster than if a PST adder were used. Furthermore, the area has also been reduced from 40 transistors down to 22 (with only 12 transistors in the first cell). This almost halves the area requirements and implies less power consumption as well.

It can also be shown quite easily that the decremter function: $a - 1 = \overline{\overline{a} + 1}$, which can be implemented with a special-purpose incrementer and an inversion of the input and output values.

6.5 Comparator structures

An adder (or more specifically, a subtracter) can also be used as a comparator. For example, to determine whether $a > b$ one can simply subtract b from a and then check if the result is greater than zero (similarly for comparing $a = b$ and $a < b$). This information is often supplied as *flags* from the arithmetic unit of a microprocessor, and can be easily deduced from the sign of the result and a post-processing “test-for-zero” (logical *nor*).

For a non self-timed adder, in which the worst case propagation delay must be accounted for, there is no major drawback in using it as a comparator. Although a special-purpose design may be slightly faster, the additional area usage is often unjustified. However using a self-timed adder as a comparator can result in a severe performance deficit. As an example, consider the comparison for $a > b$, where a and b are similar in magnitude (say, $a = 31$ and $b = 28$). The adder will then have to subtract 28 from 31, which results in the following arithmetic operations:

	0...0001 1111	31
+	1...1110 0011	<u>28</u>
+	0...0000 0001	1
<hr/>		
	0...0000 0011	sums
<hr/>		
	1...1111 1111	carries
	p...pppp ppgg	generated (g) or propagated (p)

Clearly the performance of this comparison is almost the same as the worst case performance of the adder, since all of the higher-order bits have to propagate a carry value. This type of operation occurs frequently in software programs, particularly for short loops and logical comparisons (true or false). In [Gar93] it was shown that, for the Dhrystone benchmark on the ARM processor, almost 50% of all data processing operations involving the adder required carry chains of over 28 bits in length!

A self-timed adder therefore is generally unsuitable for use as a comparator because of these long carry chains, and it is therefore necessary to implement a special-purpose self-timed comparator.

6.5.1 Possible implementations

A comparator *must* operate by comparing corresponding bits of a and b from the MSB to the LSB. If a and b are signed, then the sign bit must first be compared, and if equal, the comparator initiated. One possible implementation then is to simply start from the MSB and progress down the operand width comparing each pair of bits, and stopping as soon as a difference is detected. However, this presents no improvement in detection time for when the bits only differ at the low end of the operand: a frequent occurrence as already explained.

One method of speeding this up is to use a “forward-backward” comparator. In this implementation the comparison is also initiated from the LSB upwards, with a bit indicating which of the two operands has thus far been detected as the greatest. If the forward comparison (from the MSB downwards) hasn’t detected a difference when it coincides with the backward comparison, then the result of the latter is used. This approach approximately halves the worst case propagation time by effectively “precomparing” the lower half of the operand. The disadvantage of this approach is that even a one-half improvement is still approximately 16 bits in length (for a 32 bit operand), and the logic required for its implementation is considerably more complex.

The concept of precomparing can be extended. One approach is to use, for example, eight 4 bit subtracters which operate on adjacent slices of the operands, and to then initiate a simple ripple comparator based on the eight results. The logic requirements of this implementation are rather simple, and the worst case comparison time is reduced to

a 4 bit add and an 8 bit comparison (which has a latency somewhat less than a 12 bit carry propagation).

A better approach however is to extend the idea of precomparing down to an atomic 2 bit comparison, and reduce these through a successive tree of identical units to arrive at the result. This has the benefit of reducing the worst case comparison path to $\log_2 n = 5$ atomic units for a data width of $n = 32$ bits. However the best case comparison is now increased to this same length which then defeats the purpose of self-timing. The following section explains this comparator tree structure, and how the best case result can be improved using an asymmetric tree.

6.5.2 Comparator tree

A self-timed 2 bit comparator node can be implemented with the following equations:

$$v_{out} = v_1 + e_1 \cdot v_0$$

$$g_{out} = g_1 + e_1 \cdot g_0$$

$$e_{out} = e_1 \cdot e_0$$

where v indicates that the comparison at this node is valid, g indicates that $a > b$, and e indicates that $a = b$ thus far, with all inputs to the first row of 2 bit comparators initially low. The node to input 1 must be of a higher order than the node to input 0.

Since all input signals are initially low, so too are the outputs. At any stage v or e , but not both, could go high after activation. If $v_1 = 1$ then it is known that the preceding node has completed, and its result is passed out of this node immediately since it is of a higher order than input 0 (this is in conjunction with g to indicate which input is the greater). If however $e_1 = 1$, then the upper tree has computed an equality of the operands. The result of the lower tree then ($v_0 = 1$ or $e_0 = 1$) will be passed out to the next stage as either v_{out} or e_{out} . Note that even if input 0 computes first, the result of the comparison is not known until input 1 computes.

A dynamic logic implementation of the 2 bit comparator node is given in Fig.6.8a, and the circuitry for generating the initial inputs to a node from the operands a and b is given in Fig.6.8b.

From this 2 bit comparator structure a full 32 bit comparator can be constructed using a binary tree. This configuration is shown in Fig.6.9a, in which each circular node

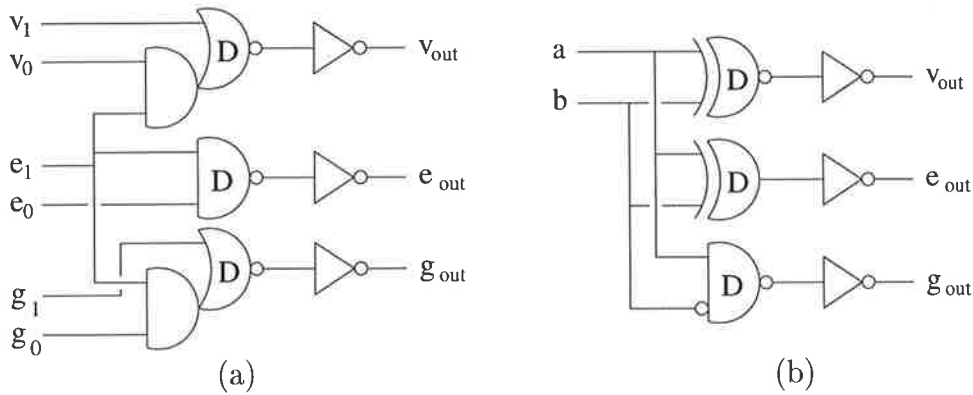


Figure 6.8: (a) An implementation of a 2 bit comparator node, and (b) the generation of the initial inputs.

represents a 2 bit comparator. However as mentioned in the previous section, this results in the best and worst case comparisons propagating through the same number of nodes, and therefore exhibiting similar computation times (although the best case will in fact be slightly faster since the pull-down node for v_1 of the 2 bit comparator requires 2 transistors, rather than 3 for v_0).

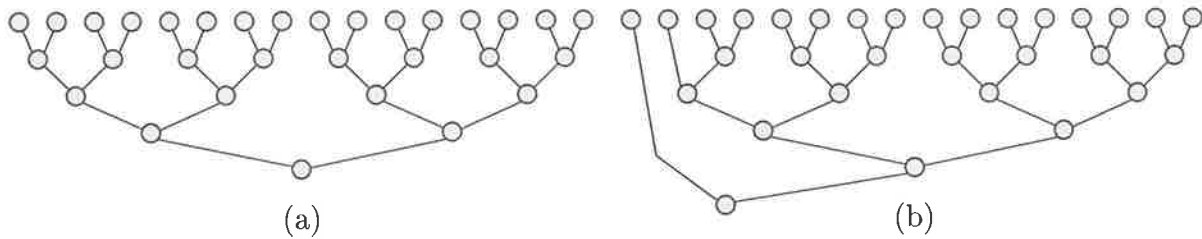


Figure 6.9: (a) A symmetric and (b) an asymmetric tree structure for a full comparator.

To enable the best case scenario to be improved, an asymmetric tree structure such as that shown in Fig.6.9b can be used. In this instance, the best case (when the MSBs are different) comparison only propagates through 2 comparator nodes, although this is now at the cost of increasing the worst case comparison to a 6 node path. Whether or not this increase is detrimental to the overall system speed depends on the frequency of low order versus high order comparisons.

It is tempting to effectively “flip” the tree structure shown in Fig.6.9b to give only a 2 node propagation path for the low-order comparison (which is frequent), in the hope that its latency will be improved. However this does not happen, since even if the low-order comparison is ready, it must *still* wait for the upper tree to compute before its result can be passed out. A speed reduction therefore results, since a 6 node propagation path must always be incurred.

6.5.3 Comparator performance

The simulation results of the comparator tree (implemented using the ES2 technology with the same design parameters as the preceding sections) are shown in Table 6.10 for both the symmetric and asymmetric tree structures. The best and worst case times are for bit variations at the MSB and LSB respectively, and the “bit 16” time is for when a difference is first detected at the 16th bit.

Comparator Design	Delay (ns)			
	best	bit 16	worst	equal
Symmetric	2.3	3.1	3.3	3.2
Asymmetric	1.1	3.5	3.9	3.8
PST Adder	9.7	5.6	9.7	9.9

Table 6.10: Simulation results of a 32 bit comparator tree.

By using an asymmetric tree structure the best case comparison time has been improved by 52%, however all other comparison times have subsequently been increased (due to an extra comparator node) by up to 19% (for when $a = b$). Since the worst case scenarios are most frequent for a comparator, the symmetric implementation will probably give the best overall performance.

It is interesting to note that although in the symmetric structure all comparisons pass through the same number of nodes, there is still a significant variation in computation time (up to 30% of worst case). The reason for this has already been stated as being due to the 2 transistor pull-down time of the dynamic nodes for a best case operation. It is also interesting to note that it’s marginally quicker to detect when $a = b$ rather than when a and b differ only in the LSB. This is again due to the faster implementation of the equality path which has less capacitance on the output of the dynamic *nand* gate.

If a PST adder were used instead as the comparator, then the comparison times of Table 6.10 would result, each corresponding to propagation delays of 31, 15, 31, and 32 bits respectively (with the first value being the worst possible in a range from 1 to 31 bits). In all circumstances the special purpose comparator gives significantly better performance, with best case improvements of 76% and 89% for the symmetric and asymmetric comparator trees respectively.

6.6 Multiplier structures

One of the fastest approaches to implementing a multiplier is to use a carry-save array [MM82], in which a 32 bit multiplier is constructed from an array of 32x32 gated full adder cells, each computing the functions:

$$\begin{aligned} pp_{i,j} &= \overline{a_i \cdot b_j} \\ sum_{i,j} &= sum_{i+1,j-1} \oplus pp_{i,j} \oplus carry_{i,j-1} \\ carry_{i,j} &= sum_{i+1,j-1} \cdot pp_{i,j} + carry_{i,j-1} (sum_{i+1,j-1} + pp_{i,j}) \end{aligned}$$

for bit i in row j . The lower 32 bits of the multiplier product emerge from the *sum* outputs of bit 0 in each stage, and the upper 32 bits of the product are computed with a vector-merging adder, which simply adds the *sum* and *carry* outputs of the last stage. Alternatively, a triangular array of gated full-adder cells could be used, although this increases the latency of the array.

The carry-save approach is inherently parallel and can be pipelined at each stage. This removes any carry propagations until the final adder is used, so there is no speed advantage in using self-timed techniques for the carry-save array, however the latency of the design could be improved slightly by self-timing the vector-merging adder (although this then increases the cycle time). Asynchronous carry-save multipliers designed for high throughput have been reported in [SK93, ML93]

Although the carry-save approach enables deep pipelining and therefore high throughput, its latency is significant. This is because 32 pipeline stages are needed in the array as well as pipelining of the adder to match the throughput requirements. Self-timing is only of benefit with regards to improving latency often at the *cost* of throughput, and therefore a carry-save multiplier is unable to take advantage of self-timed techniques.

Another popular multiplier structure is the Wallace tree [Wal64]. This approach begins with a set of 32 (or 16) partial products (PPs, which are discussed in the Section sec:ppsimpngen) and reduces these to just 2 PPs which are once again added to form the final product. The process of reduction involves taking three inputs at bit i and passing these through a full-adder cell (otherwise called a 3:2 compressor) to produce an output (*sum*) of order i and another (*carry*) of order $i + 1$. By chaining these 3:2 compressors in an appropriate way the initial set of PPs can eventually be reduced to the two required

for the final adder. Although this design approach is quite irregular it requires less logic than the carry-save technique, since fewer compressors are used to compute the upper and lower bits of the PP set.

Once again however the reduction to two PPs is an inherently parallel operation, which can have no performance gain through self-timing (except for a latency reduction in the final adder, which would again limit the throughput if the array of compressors were pipelined).

It is evident that the algorithms currently used to produce fast multipliers are not suited to self-timing, and it is therefore worth while considering an alternative implementation which enables self-timed logic to be used, in the hope of improving the multiplier latency.

6.6.1 Exploiting self-timed operation

Essentially the reduction techniques thus presented postpone the addition operation to the final stage, since for synchronous applications using a tree of such units to add the PPs is both time and area intensive (the adder structures required are up to 64 bits long and therefore need to be large and complex to provide for a sufficiently fast throughput). Ironically, this may be the *best* approach to take for implementing a self-timed multiplier, since the full data-dependency of the operation can then be exploited.

Furthermore, since the PST RC adder is small and simple, the area overhead in implementing a tree of such units is significantly less than for a synchronous approach. This could be further reduced by implementing fewer PST adders and multiplexing in previously computed PP additions, although this then increases the multiplier's latency.

Before investigating the overall implementation of such a multiplier structure, it is necessary to determine how the initial set of PPs can be generated.

6.6.2 Simple PP generation

Given that two input operands X and Y , each 32 bits long, are to be multiplied, it is possible to generate a sequence of 32 PPs by using the simple approach of multiplying each bit of X by Y . For example, assuming only a 4 bit width with $X = 0101$ (5) and $Y = 1101$ (13), a set of 4 PPs can be produced as:

						1	1	0	1	PP0	$X_0 * Y$	=	$1 * Y$
						0	0	0	0	PP1	$X_1 * Y$	=	$0 * Y$
						1	1	0	1	PP2	$X_2 * Y$	=	$1 * Y$
						0	0	0	0	PP3	$X_3 * Y$	=	$0 * Y$
0	1	0	0	0	0	0	0	0	1	R	$\sum_{i=0}^3 2^i PP_i$	=	$5Y$

Note that each PP is actually shifted according to the bit location of X before the addition. A self-timed adder tree could then be implemented with the first row of 2 adders computing $Q0 = PP0 + 2 * PP1$ and $Q1 = PP2 + 2 * PP3$, and the second row computing the result as $R = Q0 + 4 * Q1$. Note that although the result is 8 bits wide, the first row of adders only needs to be 4 bits wide (with the LSB of the result passing out directly as $PP0_0$, and the MSB emerging as the carry output).

The problem with this simple approach is that it doesn't handle signed two's complement numbers, and produces a number of PPs equal to the bit width n . If this can be reduced, then the number of adders required ($n - 1$) can also be reduced which would improve the speed and area of the design.

6.6.3 Radix 4 Booth encoding for PP generation

A radix 4 Booth's algorithm [Boo51] for producing the PPs is given in Table 6.11. Whereas in the previous encoding only 1 bit of the X operand was examined to determine the PP, in this instance 3 bits are used in an overlapping fashion to produce a range of PPs from $-2Y \rightarrow 2Y$. Bit i is paired with bit $i-1$ (with i even, and bit -1 set to zero) to determine the magnitude of the PP, and bit $i + 1$ is used to determine the sign offset (0 or $-2Y$).

X operand bits			PP
X_{i+1}	X_i	X_{i-1}	
0	0	0	0
0	0	1	Y
0	1	0	Y
0	1	1	$2Y$
1	0	0	$-2Y$
1	0	1	$-Y$
1	1	0	$-Y$
1	1	1	0

Table 6.11: Radix 4 Booth's algorithm for encoding the PPs.

To illustrate this procedure, consider again the example of the previous section with $X = 0101$. In this instance, the first triplet of X to be considered to produce $PP0$ is $X = 010$ (bit locations 1, 0, and a zero extension), and the second triplet to produce $PP1$ is also $X = 010$ (bit locations 3, 2, and 1).

	1	1	0	1	PP0	$X_{1,0,-1} \rightarrow 010$	=	Y
	1	1	0	1	PP1	$X_{3,2,1} \rightarrow 010$	=	Y
0	1	0	0	0	0	0	1	R
						$\sum_{i=0}^1 2^{2i} PP_i$	=	$5Y$

The radix 4 algorithm therefore halves the number of PPs required for signed 32 bit operands to 16 rather than 32. Generating the PPs of value $2Y$, Y , and 0 is trivial (simply shifts or masking of the input operand Y), however to generate $-Y$ and $-2Y$ requires a two's complement inversion which involves using an incrementer (since $-Y = \overline{Y} + 1$).

In a Wallace tree architecture the incrementing function can in fact be implemented with additional compressors for the input carry. For an adder tree a similar technique can be used, however at best there will still need to be an incrementer after the final adder. To illustrate why this is, consider the case in which $PP0 = -Y$ and $PP1 = -2Y$. The result of the addition $Q0 = PP0 + 4PP1$ will need to compute the function: $Q0 = \overline{Y} + 4(\overline{2Y} + 1) + 1$, which clearly involves two carries at bit locations 0 and 2, whereas the PST adder structure allows for only one input carry. Since the lower two bits of the addition can be passed out directly, the carry at bit location 2 is applied to this adder, and the other carry has to be postponed to the following stage. All PPs can have their carry postponed to a subsequent adder stage if necessary except for $PP0$ (since there are 8 possible carries from the first layer and only 7 subsequent adders), which therefore has to use an incrementer after the final adder. Since a self-timed incrementer is typically fast the increase in latency is small. Furthermore this delay is only incurred when $PP0$ is negative (for 37.5% of cases) whereas it would almost always be incurred if used to generate negative PP values to the first adder stage.

A self-timed adder computes in the fastest time when both of its operands are equal. For the Booth encoding of Table 6.11 this can only occur when $PP0 = PP1 = 0$, which happens only 6% of the time for random data. The following section describes a technique in which the Booth's algorithm is *recoded* to give a 22% frequency of best case adder performance, and also reduces the frequency of use of the final incrementer to just 3% of cases.

6.6.4 Recoding Booth's algorithm to improve performance

Given the Booth encoding of Table 6.11, it is a simple matter to determine the range of outputs from the first adder stage of $Q_0 = PP_0 + 4PP_1$ (and similarly for $Q_1 \dots Q_7$), as given in Table 6.12. Note that for each value of PP_0 there can only be 4 possible values of PP_1 , since adjacent triplets overlap at the central bit (as indicated by the boxed value in the table).

PP0 (*Y)	Possible PP1 values (*Y)				Q0 (*Y)			
000 = 0	000 = 0	010 = 1	100 = -2	110 = -1	0	4	-8	-4
001 = 1	:	:	:	:	1	5	-7	-3
010 = 1	:	:	:	:	1	5	-7	-3
011 = 2	:	:	:	:	2	6	-6	-2
100 = -2	001 = 1	011 = 2	101 = -1	111 = 0	2	6	-6	-2
101 = -1	:	:	:	:	3	7	-5	-1
110 = -1	:	:	:	:	3	7	-5	-1
111 = 0	:	:	:	:	4	8	-4	0

Table 6.12: Combinations of PP_0 and PP_1 which produce an output sum Q_0 .

It is clear from this that the range of values for Q_0 is $-8Y \rightarrow 8Y$, and by considering both triplets for PP_0 and PP_1 together (that is, 5 bits of X at a time) it is possible to recode the PP values to improve the adder's performance, and yet still produce the correct output sums for Q_0 . For example, the combination of $PP_0 = 0$ and $4PP_1 = 4Y$ (for $X_{5bits} = 01000$) can be recoded to give $PP_0 = PP_1 = 2Y$. As required the sum is unchanged ($4Y$) however by recoding both PPs to be identical the self-timed adder will now compute in the fastest time regardless of the value of Y . All summations resulting in $Q_0 = 0, 2Y, 4Y$, and $8Y$ can be thus recoded, resulting in 7 out of 32 cases (22%) computing in the fastest time. Note that PP_0 and PP_1 are now of equal weight ($Q_0 = PP_0 + PP_1$) which requires the first stage adders to be of the same bit width as the PPs.

Furthermore, recoding can reduce the incidence of carry propagations to the subsequent incrementer. For example, the combination of $PP_0 = -2Y$ and $PP_1 = -Y$ (for $X_{5bits} = 10100$) can be recoded to give $Q_0 = 2Y - 8Y = -6Y$. This can now be implemented with an input carry of one to the first stage and prevents the need for a carry of "1" to propagate through to the next stage, and eventually to the incrementer, as would otherwise be needed. Table 6.13 gives the PP recoding used in the self-timed multiplier design for each X quintuplet.

X_{5bits}	PP0	PP1	X_{5bits}	PP0	PP1	X_{5bits}	PP0	PP1	X_{5bits}	PP0	PP1
00000	0	0	01000	2Y	2Y	10000	0	-8Y	11000	0	-4Y
00001	0	Y	01001	Y	4Y	10001	Y	-8Y	11001	Y	-4Y
00010	0	Y	01010	Y	4Y	10010	Y	-8Y	11010	Y	-4Y
00011	Y	Y	01011	2Y	4Y	10011	2Y	-8Y	11011	2Y	-4Y
00100	Y	Y	01100	8Y	-2Y	10100	2Y	-8Y	11100	0	-2Y
00101	Y	2Y	01101	8Y	-Y	10101	Y	4Y	11101	0	-Y
00110	Y	2Y	01110	8Y	-Y	10110	Y	4Y	11110	0	-Y
00111	2Y	2Y	01111	8Y	0	10111	0	-4Y	11111	0	0

Table 6.13: Recoding of Booth's algorithm to provide the PPs.

All values for Q_0 in the range $-8Y \rightarrow 8Y$ can be recoded to remove any carry propagation to the next stage, except for $Q_0 = -5Y = \overline{5Y} + 1$. For this case (shown boxed in the table) a value of $5Y$ is computed and inverted, and a carry then propagated to the following stage. Since for PP0 this can only happen when $X = 10110$ (because a zero extension is applied) the final stage incrementer will be used in just 3% of cases (1 in 32 times).

Given the recoded requirements for PP0 and PP1 it's possible to produce from X_{5bits} various control signals which multiplex in the appropriate bit-shift of Y , perform masking, inversion, and issue the carry in and carry propagate signals.

6.6.5 Implementation, floorplanning, and area usage

Figure 6.10 shows the overall implementation of the multiplier, indicating the adder widths required (in the shaded boxes), their relative bit positions in the product generation, and the bypassing of data values. For such a structure it is important to consider how it should be floorplanned in VLSI. Two possible approaches are shown in Fig.6.11.

The floorplan of Fig.6.11a has a breadth governed by the first layer of eight adders. The adders of the second layer are placed beneath this, and those of the last 2 layers are slotted into the remaining gaps. Due to the increasing width of the adders this second array is approximately the same width as the first.

The floorplan of Fig.6.11b simply flips the upper half of the adder array beneath the lower half, and implements the extraneous 48 bit adder and 64 bit incrementer to the right of this. This approach gives an aspect ratio much closer to one, which may be required in an embedded application.

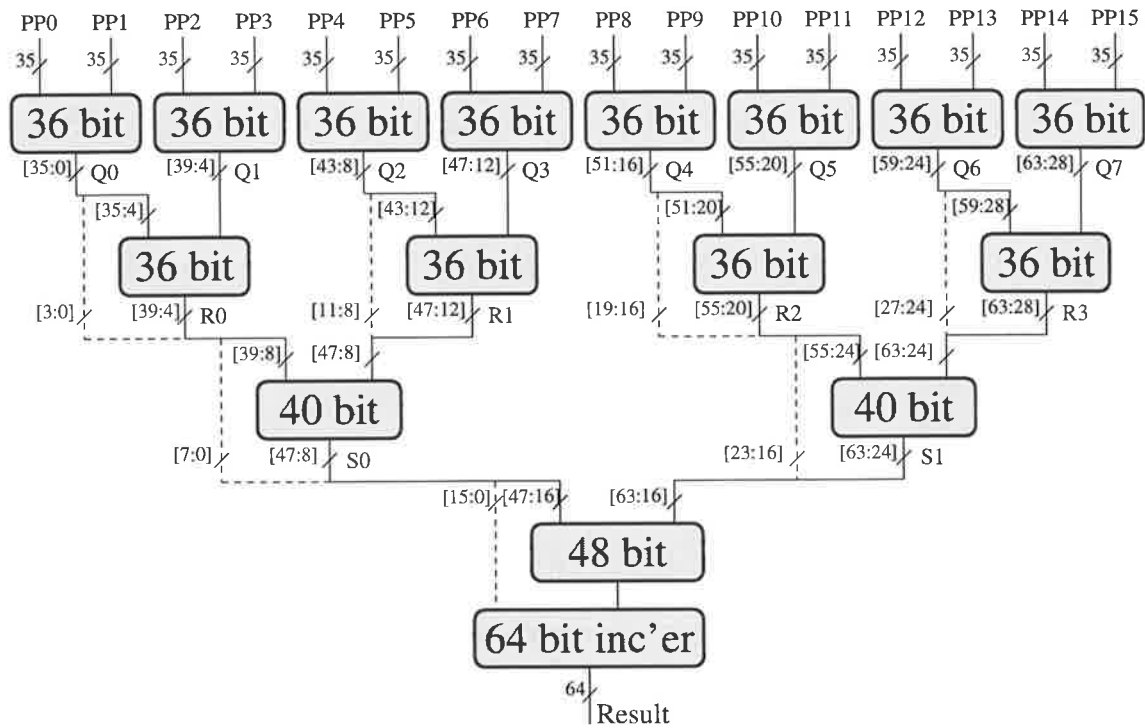


Figure 6.10: Configuration of a self-timed multiplier.

The multiplier has been implemented in the ES2 technology using PST adders and a special-purpose incrementer. The floorplan of Fig.6.11a results in an area of $6.42mm^2$ with an aspect ratio of 6.2, and that of Fig.6.11b results in values of $7.12mm^2$ and 1.7 respectively.

6.6.6 Performance and comparisons

The self-timed multiplier has been simulated using the switch-level simulator IRSIM with a parameter file designed to give the closest approximation to various Hspice (level 13) simulations of smaller circuits. The results of this simulation are shown in Table 6.14 together with the best known asynchronous and synchronous CMOS signed integer multipliers [SK93, YYN⁺90]. For the self-timed ECS multiplier, the best case multiplication occurs for $0*0$, and the worst case occurs for $1*-1$. The average case time was determined from a long test sequence of multiplications with random data.

The 4P LDPL design [SK93] implements a 16 bit multiplier in a $1.0\mu m$ technology, and uses a pipelined carry-save array and a pipelined Manchester carry adder to compute the result. The synchronous design [YYN⁺90] also implements a 16 bit multiplier in a $0.5\mu m$ technology, but uses a Wallace tree and carry lookahead adders to compute the result.

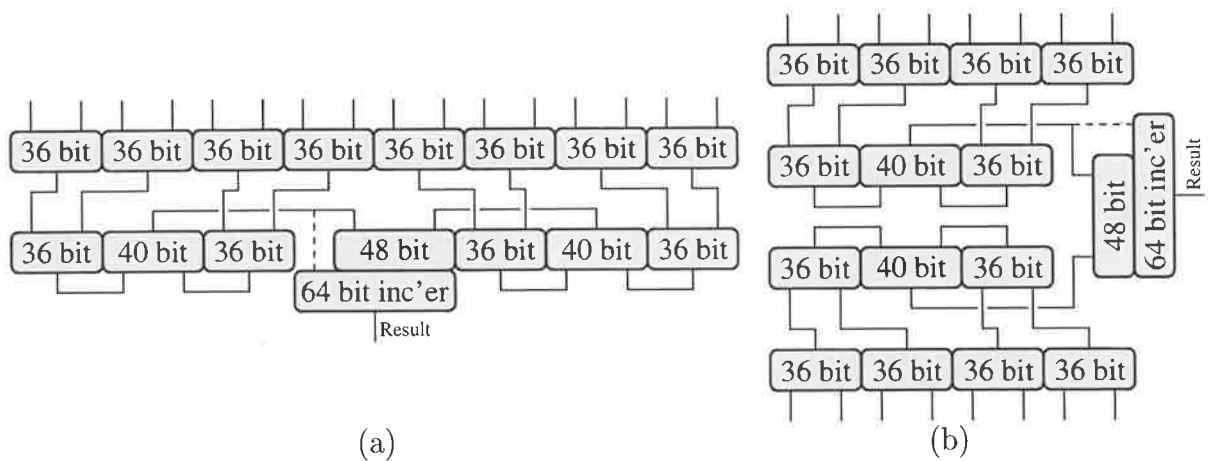


Figure 6.11: (a) A possible floorplan for a self-timed multiplier, and (b) a floorplan with an aspect ratio closer to one.

Multiplier Design	Latency (ns)			Throughput (ns)	Area (mm^2)	Density (T/mm^2)
	best	ave	worst			
ECS self-timed	17.2	21.5	47.6	latency+3	6.42	7300
4P LDPL		59.6		3.2	7.25	??
Synchronous		10.4		10.4 + ??	22.6	1500

Table 6.14: Simulation results and comparisons of a 32 bit signed integer self-timed multiplier.

Since these designs are implemented in different technologies than the ECS design, their results have been scaled as described in Section 6.3.7 to correspond to a multiplication in a $0.7\mu m$ technology to enable a more accurate comparison.

To scale the synchronous design to a 32 bit data width, its latency has been increased by 40% which accounts for the increase in Wallace tree depth (from 6 layers to 8, giving a 33% increase) and an increase in the final adder's latency. For the LDPL design, 18 extra stages have been added to the pipeline (for which 20 were used in the 16 bit implementation) to account for 16 extra multiplier array stages and 2 extra carry propagate stages. Both designs have had their area (and number of transistors) scaled by a factor of 4 due to the quadratic increase in the array size.

Interestingly, both of these designs use complementary pass transistor logic, in which dual rail signals are computed from two complementary nmos pull-down trees (as mentioned in Section 5.3.1), but with their sources connected to the input signals (as well as their gates) rather than to ground through an activate transistor. This can however result in significant power dissipation during precharging and a slow *serial* precharge time (since at precharge all pull-down paths will initially be on).

The ECS multiplier has a significantly lower latency than the LDPL design even in the worst case, with a best case improvement of 71%, and the area usage has also been reduced. The cycle time of the LDPL design is of course faster since a deep pipeline has been used, whereas the ECS multiplier has none. As previously stated, self-timing is not beneficial in improving the throughput over non self-timed structures (only the latency can be improved), a fact acknowledged in [SK93] in which a corresponding synchronous design gives a faster throughput of 2.9ns (scaled).

In contrast the multiplier in [YYN⁺90] is not pipelined and results in a better latency than the ECS multiplier even in the best case scenario, however its area usage is substantially more. Given that the transistor density of this design is almost 80% less, it could be expected that significantly larger transistor widths have been used to obtain the high throughput (although wiring and irregularity in the Wallace tree structure will also contribute to the density difference). Resizing the transistors in the ECS multiplier could potentially produce a typical latency closer to that of the synchronous design whilst still occupying less area. Since the precharge time of the synchronous design is not quoted its effect on the cycle time is unknown.

6.6.7 Potential improvements

The ECS self-timed multiplier as presented could be further improved by implementing the following structural and algorithmic modifications.

6.6.7.1 Area reduction

To approximately *halve* the area usage it is possible to re-use the first layer of 8 adders to implement the additions which would otherwise be required in the subsequent layers. This can be done with a minimal effect on the latency, since an adder in the second layer cannot be activated anyway until its two source operands from the first stage adders are available. As such the only possible delay incurred will be in waiting for the re-used adder to precharge, which can occur whilst the new source operands are multiplexed in. Note also that since multiplexers are already used at the inputs to this layer, the overhead in introducing another multiplexer signal is negligible. Fig.6.12 shows such an implementation, with the first set of adder inputs given by the thick inner wires and the

second by the thin outer wires, and the first set of adder outputs given by the solid wires and the second set by the dashed. Reducing the number of adders any further introduces additional delays since the first stage additions will have to share a common unit.

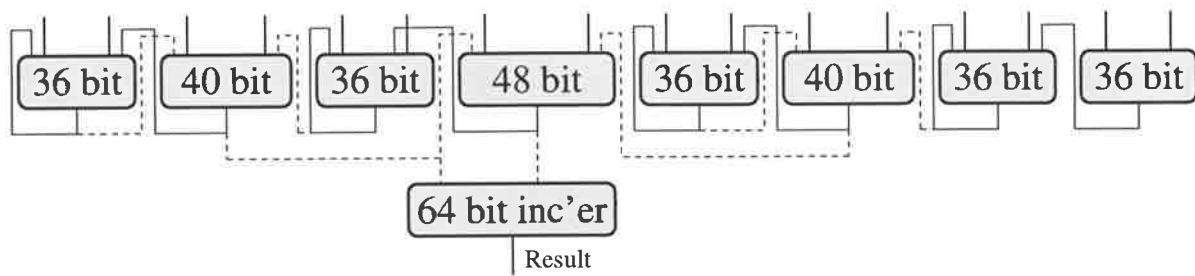


Figure 6.12: A low area implementation of a self-timed multiplier.

6.6.7.2 Cycle time reduction

Similarly, since each layer of the multiplier must wait for the previous layer to compute, these layers can be *pipelined* to increase the throughput. This is not in contradiction to previous statements regarding self-timing and pipelines since the partitioning is placed *between*, and not *within*, the self-timed units. The average throughput then will be governed by the maximum of the average latency of any one layer, which can be determined from the statistical distribution of a single adder in [Gar93] as being: 7.6, 6.7, 6.0, and 5.0 bits for the first through last layers respectively. Therefore a cycle time of about 10ns could be expected with an ECS $PP\beta$ pipeline controller (assuming a 3ns precharge time, 2ns driver time, 3.2ns computation time from Table 6.6, and a control overhead of 1.8ns from Table 5.6), with a small increase in latency of about 2ns (0.5ns per stage from Table 5.6). Note that implementing this optimization precludes the area reduction technique previously mentioned.

6.6.7.3 Latency reduction

Aside from transistor and layout optimizations to improve latency, there is a simple algorithmic improvement which could also be made. The recoding in Table 6.13 enables 6 of the 32 operations (19%) to compute in the minimum time, however it can also be observed from Table 6.12 that half (50%) of the sums for Q_0 have a magnitude equal to either a bit shift or masking of the Y operand. For these instances the sum can be multiplexed directly from Y (and conditionally inverted for a negative value) without

having to activate the adder at all! A minor drawback is that 9 in 32 operations (28%) would now require the final incrementer. Implementing this optimization would result in a significant power saving as well as improving the latency (and the cycle time if pipelined) by about 1ns (again computed from the statistical distribution in [Gar93]).

The latency of the incrementer can also be removed for almost *every* operation. Since the lower 16 bits of the result are available before the final 48 bit adder is activated, an increment on these bits can be initiated in parallel with the addition. In the majority of cases this will complete before the final adder, and will not need to be activated on the adder's result unless a carry from bit 15 is generated (a very rare occurrence).

6.7 Summary

This chapter has presented a range of fast self-timed adders and various other derivatives which are typically faster, less power consuming, and smaller than any that have been previously reported. This enables various microprocessor operations such as branch target calculations, branch detection, and PC incrementing to be implemented quickly without the need for complex logic structures. For these kind of operations, a low computation latency is especially desirable.

A majority of programs also exhibit a significant amount of data dependency between subsequent (or nearby) instructions. In such instances enabling low latency execution through self-timing could reduce the stall time of the following instruction, and hence increase the processor speed. Although the throughput of self-timed units is less than could be achieved by pipelining, employing a degree of parallelism (having more than one such unit) could overcome this problem. It may therefore be construed that a superscalar processor structure is naturally suited to self-timed computations, whereas in contrast, BD computational units would be ideally suited to a pipelined processor structure. Such an investigation is embodied in the following two chapters.

Chapter 7

ECSTAC: A Pipelined Microprocessor

AN asynchronous microprocessor called *ECSTAC* (**E**vent **C**ontrolled **S**ystems **T**emporally specified **A**synchronous **C**PU) has been designed using a predominantly pipelined architecture. Although asynchronous pipelined processors are not new, the goals behind the design of *ECSTAC*, and the techniques used in its implementation, are sufficiently different to those of other processors to justify the new design.

In particular, designing a microprocessor is an extremely complex task involving a plethora of control and data interactions. Such a complex system enables the potential of ECS as a design framework to be properly examined, since the investigation of the comparatively small sub-systems thus far merely hints at this, despite having demonstrated significant advantages at this level. Furthermore, since other asynchronous microprocessors have been developed, an accurate and reliable comparison of ECS can then be made. Issues such as design area, instruction execution rate, power dissipation, etc. can all be quantitatively measured to support such a comparison.

Another reason for undertaking the design of a complex microprocessor was to provide a vehicle for the creation of new ideas and techniques. As new design problems arose so too did new solutions, and out of this progressive development have arisen many of the design approaches and methodology issues presented in the preceding chapters. In particular this development has led to the refinement of the ECS representation, early implementations of the ECS pipelines and PST structures, and the majority of fast asynchronous circuit techniques.

Since the ECS paradigm is geared towards implementing high speed control structures, it is logical that an example system be chosen which needs to operate at high speed, and

yet is still challenging enough to provide some difficult design problems. Microprocessor design has invariably been driven by the need for high speed operation, and as such it becomes an obvious choice. This is further supported by the fact that low power consumption is a secondary concern of ECS, and is similarly an important (yet secondary) topic for most microprocessors. The issue of whether or not low power consumption still results from a paradigm geared primarily for high speed can then also be determined.

There is lastly the issue of architectural alternatives, although this was not of concern at the inception of *ECSTAC* but developed during its fruition. *ECSTAC* is constructed as a pipelined processor not unlike the majority of early synchronous designs, however as new information unfolded, it became apparent that perhaps the best method of exploiting asynchrony in a microprocessor was to implement a superscalar structure. Therefore if the latter were also implemented then a comparison could be made between both architectural approaches, and a direction for future asynchronous microprocessors might result.

7.1 Design considerations

The cost of fabrication restricted the amount of silicon area which could be used for the processor, and this in turn resulted in some restrictions being placed upon the architecture. Firstly, to keep the processing area down only an 8 bit data path was used, which is reasonable since the processor's operating speed is governed almost entirely by the event control and would therefore be only marginally different to a 32 bit implementation. However, to enable sufficiently complex program codes to be compiled it was decided to use a 24 bit address path. In retrospect it is clear that this mismatch of data and address widths seriously affected the resulting performance by complicating numerous critical control structures, and it would have been more practical to have these be identical. Furthermore, since an on-chip cache was implemented (to facilitate high speed instruction issue) and is relatively much larger than the processor core, implementing a full 32 bit data and address path would not have been significantly more area intensive than first thought. Unfortunately since these factors were not realized until after the design was well under way, a 24 bit address and 8 bit data path remained.

Due to the expected high speed of the processor it was deemed necessary to incorporate an on-chip cache, as already mentioned. Separate instruction and data caches were

employed since a combined cache requires a significant amount of arbitration between instruction fetching and data operations, which could then limit the processor's speed. This work was undertaken by Sam Appleton, and is not described in any detail in this thesis since the author's contribution was in the architecture of the processor core. Specifics of the cache design can be found in [AML95a, App96]. The core simply interprets both caches as *black box* memory units.

To simplify the design there were no interrupt or exception handling facilities incorporated. Nor was there any provision for floating point operations, since these seem rather pointless in a prototype 8 bit machine. Only integer operations are supported by the core, excluding integer multiplication and division.

Sixteen general-purpose 8 bit registers were provided as well as a flags register (FR), and a 24 bit stack pointer (SP) and program counter (PC). A custom RISC-like ISA was implemented to enable the complexity of the design to be adequately managed (although this then complicates the issue of performance comparisons).

7.2 Instruction set architecture (ISA)

Only an overview of the ISA for *ECSTAC* is given here, however a complete description of all instruction codes, operating modes, and bit encoding is given in Appendix B. There are a total of 47 distinct instruction types excluding mode variants, and 87 including mode variants.

The first class of instructions include the memory accessing LD and ST instructions, which retrieve and store data from and to memory respectively. Each of these instructions have three mode variants: register mode (in which the address is computed as the sum of two register quadruples); offset mode (address = register quadruple + offset); and direct mode (a 24 bit address is encoded in the instruction). The first two modes require only 2 bytes for their encoding but the third requires 4 bytes, of which the address itself uses 3 (to specify a 24 bit address width).

The dissimilarity between bus sizes results in a number of awkward situations throughout the microprocessor architecture, one of which is the fact that the modes involving register accesses in fact have to fetch data from three separate registers as opposed to only one if the bus widths were compatible. To alleviate this problem the 16 registers are

grouped into four register quadruples (Q0, Q4, Q8, and Q12) for the three byte register accesses (termed a “Qfetch”, although the last byte of the quadruple isn’t actually used), and the read port from the register bank provides three output buses. Note however that because each Qfetch starts from just one of four possible registers, the output bus locations of its three register accesses are known, and therefore the register cells themselves only require a *two* read port cell design.

The next class of instructions involve the ALU, incorporating left and right arithmetic and logical shifts, signed and unsigned additions and subtractions, four logical operations (nand, nor, xor, and not), and negation, increment, and decrement operations. The move instruction also passes through the ALU, although no processing is performed. ALU instructions may be encoded in the short mode (which requires two bytes) or the long mode (which requires three), however the former requires the destination register and one source register to be identical, whereas the latter does not. Note that all ALU instructions with only one operand (9 of the 16 available) will be encoded as *short*, since a unique source and destination register can still be specified. Two further modes are also available: register mode (specifying a source register for the second ALU operand) and offset mode (encoding a constant 8 bit value instead); however for the *short* mode encoding, the offset may only be 4 bits long instead of 8, and is considered as an unsigned constant.

The branch instructions form yet another class. The CALL instruction has a one byte encoding with the branch address contained in a register quadruple, and the RETN instruction is also encoded with one byte and reloads the PC with the last value stored in the stack (which should always be PC+1 from the calling instruction). The JUMP instructions (encoded in two bytes) allow both unconditional branches and branches dependent on any one of 14 combinations of the contents of the FR, which contains zero, parity, sign, overflow, and carry flags. Two different jump modes are also available: register mode, in which the branch address is contained in a register quadruple; and offset mode, which enables jumps of +127 or -128 locations relative to the current PC value.

Four stack operations are provided (each encoded as a one byte instruction) which enable the contents of a register (including the FR) to be PUSHed or POPped. Finally there is a group of miscellaneous one byte instructions which include NOOP (do nothing), TRSP (transfer the contents of Q12 to the SP, which enables this register to be initialized before use), HALT (to stall the processor through software), FLSH (to invalidate the

contents of the data cache), and four instructions to enable or disable the instruction and data caches.

7.3 Architectural overview

A block diagram of the microprocessor architecture is shown in Fig.7.1. The arrows between modules indicate the general data flow and the *ticks* on each module represent a row of latching elements between pipeline stages.

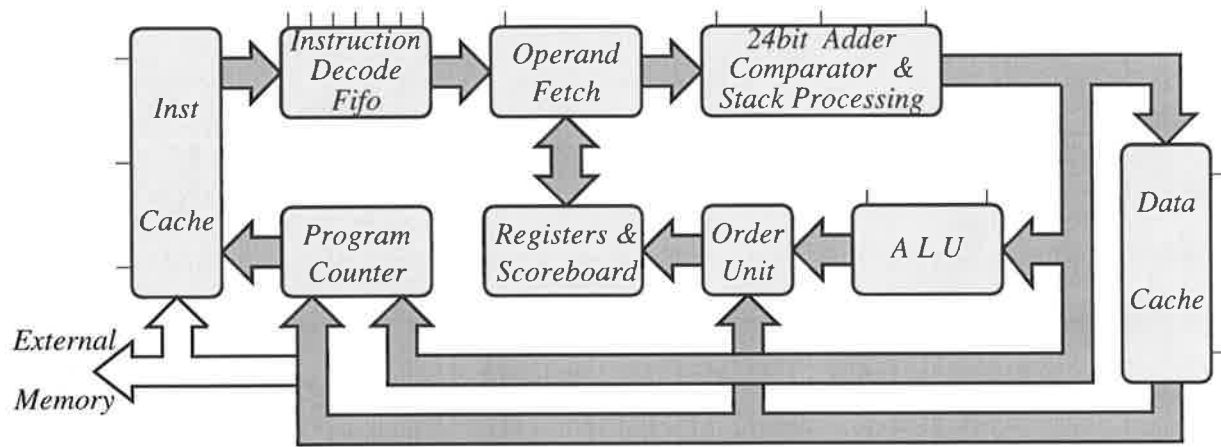


Figure 7.1: The general structure of the *ECSTAC* microprocessor.

Separate instruction and data caches (IC and DC) are used in the design to delineate between program control and data accessing, and to enable arbitration to be placed at the cache outputs to external memory as opposed to the input of a joint instruction and data cache. This significantly reduces both the arbitration frequency and the effect of metastability delays, as external memory accesses are comparatively slow anyway.

The IC continuously fetches bytes from the PC address and places the data into the instruction decode (ID) pipeline. This is heavily pipelined so that it effectively acts as a FIFO buffer (the decoding in each stage is minimal). The results then pass into the operand fetch (OF) stage which, from the information provided by the ID pipeline, controls the number of register fetches required and the organization of these (together with any immediate input values and the PC if required) onto a minimal set of output buses.

The adder, comparator, and stack processing (ACS) stage performs address offset additions (which are 24 bits long and would be awkward to compute with the ALU's

8 bit adder) in two stages employing 12 bit Manchester carry adders (MCAs) in each. The second stage (the upper 12 bit addition) is bypassed if the intermediate carry is zero (as the result of this prediction is determined in the first stage). The first stage also determines whether or not a JUMP operation is to be taken. If it's not, then this operation is converted into a NOOP (therefore bypassing any 24 bit additions that may have commenced) which then terminates at the ACS output. Otherwise, a signal is sent back to all preceding stages which converts any instructions therein into NOOPs, since they are now invalidated due to the branch. The IC-PC interaction is also instructed to halt until a new PC value has been written to it. The second stage of the ACS incorporates the SP, which places the relevant address onto the bus to the DC for writing and reading from memory. The CALL and RETN instructions present a problem here as they require the pushing and popping of the PC, which is 24 bits wide (whereas the data width is only 8 bits wide), therefore special refetch control is incorporated into this stage to effect this. At the output of the ACS stage the appropriate execution unit (DC, ALU, PC, or nothing for a NOOP) is selected.

The ALU is a single pipeline stage employing a dedicated shifter and logical unit and a dedicated 8 bit MCA. The operation time of this unit will vary depending on which of these, if any, is required by the operation. The DC is also a single pipeline stage which will generally write back to registers except for a RETN instruction which writes back to the PC.

The register bank employs a standard two read port and one write port cell design, and each register array has associated with it a corresponding tag which enables data dependencies to be properly handled (scoreboarding). When an instruction passes through the OF it attempts to read the source register and, if the tag is high, is successful. Otherwise it must wait until such time that the tag does go high. When all of the required data is read it then tags (sets low) its destination register if one exists, and then proceeds through the rest of the pipeline. Any subsequent instructions wanting to access this register will therefore be stalled until it is later written to and the tag removed. Thus there can never be simultaneous read and write operations for the same register, although for different registers this is still possible.

The PC is a simple incrementing 24 bit register, which can have data written to it from the ACS stage or the DC. Note that these can never occur simultaneously (therefore

avoiding arbitration) because the detection of a program branch in the ACS stage will convert all preceding instructions into NOOPs, and will stall all subsequent PC reads until it is re-written to. There is however a degree of arbitration with regards to halting the IF-PC interaction once a branch is detected, which is resolved using the technique of Section 4.4.1.

7.4 Processor sub-systems

This section gives a more complete description of the logic blocks of Fig.7.1, focussing on the control design techniques used to improve throughput and reduce latency.

7.4.1 Instruction decode

The ID stage receives a constant stream of 8 bit data values from the IC (as well as the corresponding 24 bit PC location of the data), which have to be decoded to provide numerous control signals for use in the latter stages, such as which functional unit to trigger, how many bytes in the instruction, how many and what registers to source, etc. It was also deemed necessary to have a FIFO buffer after the IC for two reasons. Firstly, if the IC missed then there would be a store of instructions still in the FIFO which could get executed whilst the IC fetched the next stream of instructions from external memory, and secondly so that if a latter stage stalled (due perhaps to a miss in the DC, or a refetch operation for the SP or register sourcing) then the FIFO could fill up and thereby reduce the frequency of stalls in the IC. Both issues help to maintain a steady flow of instructions through the processor core. Note however that implementing a FIFO here does have the drawback of creating a longer branch latency (from when the branch instruction is first fetched to when the branch target instruction is fetched), which can reduce performance.

Consequently, the ID process is actually combined with the FIFO, resulting in a 7 stage pipeline design. To enable the pipeline to still act as a FIFO, which requires a very low processing latency in each stage, the decoding process was reduced to a minimal tree of 2-input *nand* and *nor* gates with only one such gate in each stage. This ensured a high throughput and low latency of the ID and FIFO combination.

Note that with the ISA of *ECSTAC* the decoding process only needs to be applied to the first byte of any instruction, since any subsequent bytes if present only contain

register locations, immediate values, or function codes (which are decoded in their relevant functional unit). However to implement this requires the instruction length to be initially decoded (so that the start of the next instruction can be determined) and then de-activated for any subsequent bytes. Implementing this would complicate the control circuitry and increase the stage latency, but would decrease the power consumption. Since the former is of most importance this design alternative is not implemented, and instead every byte in the instruction stream is decoded. Only in the OF stage are the correct control signals latched from the first byte of the instruction, and for any subsequent bytes discarded.

7.4.2 Operand fetch

The OF stage sources the required register values and then tags the destination register if used. The register values are then multiplexed with the immediate operands (from bytes 2, 3, or 4) to provide the following stages with the appropriate data values. A block diagram of the structure is shown in Fig.7.2.

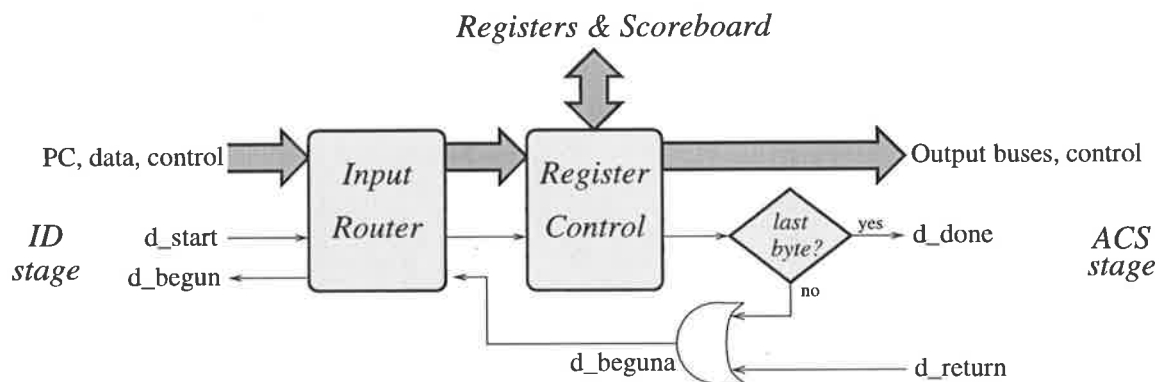


Figure 7.2: A block diagram of the OF stage.

It will be seen from this that the OF unit uses a single pipeline stage operating at the byte rate (as per the ID stage), however all subsequent stages are activated at the instruction rate. Using a single stage for the register operations removes the contention which would otherwise be present if they were spread over more than one stage. To maintain a high throughput however it is essential that the router and register control blocks (and the registers themselves) operate quickly.

The router is used to interface the OF stage to the ID, and is required to latch the control bits, PC location, and data value from the first byte of each instruction, and only

the data value for any subsequent bytes. The control circuit used for this is shown in Fig.7.3.

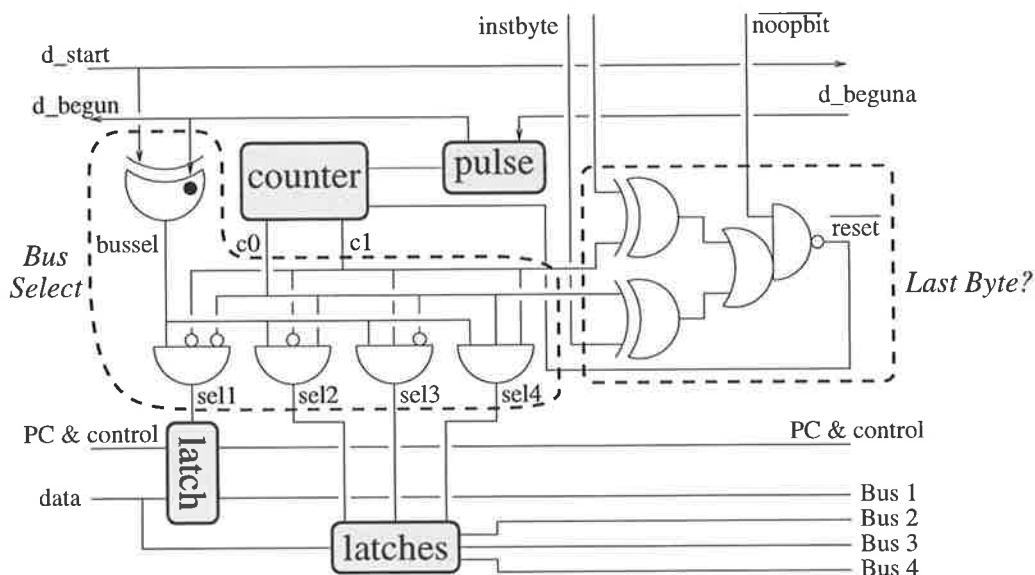


Figure 7.3: Control circuit for routing the data from the ID into the OF stage.

The 2 bit control signal *instbyte* specifies the number of bytes which are present in the instruction, which from Appendix B is between 1 and 4 in length. A 2 bit counter is used to keep track of the number of bytes which have thus far been processed, and when equal to *instbyte* gets reset to zero ready for the start of the next instruction. The $\overline{noopbit}$ signal also resets the counter for when the instruction gets converted to a NOOP (when it's in the shadow of a taken branch).

From this counter the relevant output bus for the data values can be selected, by appropriately masking the latch select signal *bussel* which is generated by the TE: $bussel \leftarrow \overline{dbeguna} \cup dstart$, so that the relevant bus data gets latched on each *dstart* event. The only latency incurred by this unit is in incrementing the counter before providing a return event back to the ID stage, since if *dbeguna* were used instead then the counter may not properly mask *bussel* before the next instruction arrives. Note that the forward latency from *dstart* is zero, since the delay from this to $\nabla bussel$ is significantly less than the delay to *dbeguna* (as will be seen by the structure of the register controller).

The register accessing requirements for each of the 16 fundamental instructions (as given by the first 4 bits of the first byte) are presented in Table 7.1, in which *Qi* specifies a 24 bit source (*Qfetch*), *Ri* specifies an 8 bit source (*Rfetch*), and *T(Ri)* specifies that register *i* must be tagged once all sourcing operations are complete.

Code	Instruction + Mode	Byte 1	Byte 2	Byte 3	Byte 4
0000	LD register	Qx	Qy & T(Rz)		
0001	LD offset	Qx	T(Rz)		
0010	LD direct	T(Rz)	-	-	-
0011	JUMP offset	FR	-		
0100	ST register	Qx	Rz & Qy		
0101	ST offset	Qx	Rz		
0110	ST direct	Rz	-	-	-
0111	JUMP register	FR	Qz		
1000	ALU short/register	-	Ry, Rz ¹ & T(Rz)		
1001	ALU short/offset	-	Rz & T(Rz)		
1010	ALU long/register	Rx	Ry & T(Rz)	-	
1011	ALU long/offset	Rx	T(Rz)	-	
1100	POP	T(Rx)			
1101	PUSH	Rx			
1110	SPECIAL	<i>special</i> ²			
1111	CALL	Qz			

1: a second register source (Rz) is only necessary for ALU operations with 2 operands.

2: a register access of Q12 is needed for TRSP, the FR for PSHF, and a FR tag for POPF.

Table 7.1: Register accessing requirements of the fundamental instruction set.

It can be seen that register operations are only ever required for the first 2 bytes of an instruction, and that for the second byte there may in fact be 2 register source operations which have to occur, as well as potentially having to tag a register. Implementing such a register controller is therefore a reasonably complex task.

One approach is to activate the register accesses using *select* and *merge* gates, as shown in Fig.7.4. The signals *go1* and *go2* are used to indicate whether one or two register sources are required for each byte (although the latter is rare), and *go0* and *go3* indicate if the FR or register bank has to be tagged. These signals are generated from control bits decoded in the ID stage and the current counter value. Since the tagging functions can be done in a fast, constant time they are triggered without acknowledgement. *Merge* and *restore* gates are used to combine the two possible source operations prior to the register bank. Note that tagging of the registers must occur *after* the register fetching and hence these operations must be serially connected, although for the FR a fetch and a tag can never occur together in the same instruction, therefore it can be initiated immediately.

This approach gives a best case latency from $\partial start \rightarrow \partial startc$ of $2(t_{sel} + t_{merge}) = 40$ using Table 3.1 as a guide, a nominal latency of $40 + t_{merge} + t_{res} = 64$ (for just one register source), and a worst case latency of $64 + t_{merge} + t_{res} = 88$ (for two register sources),

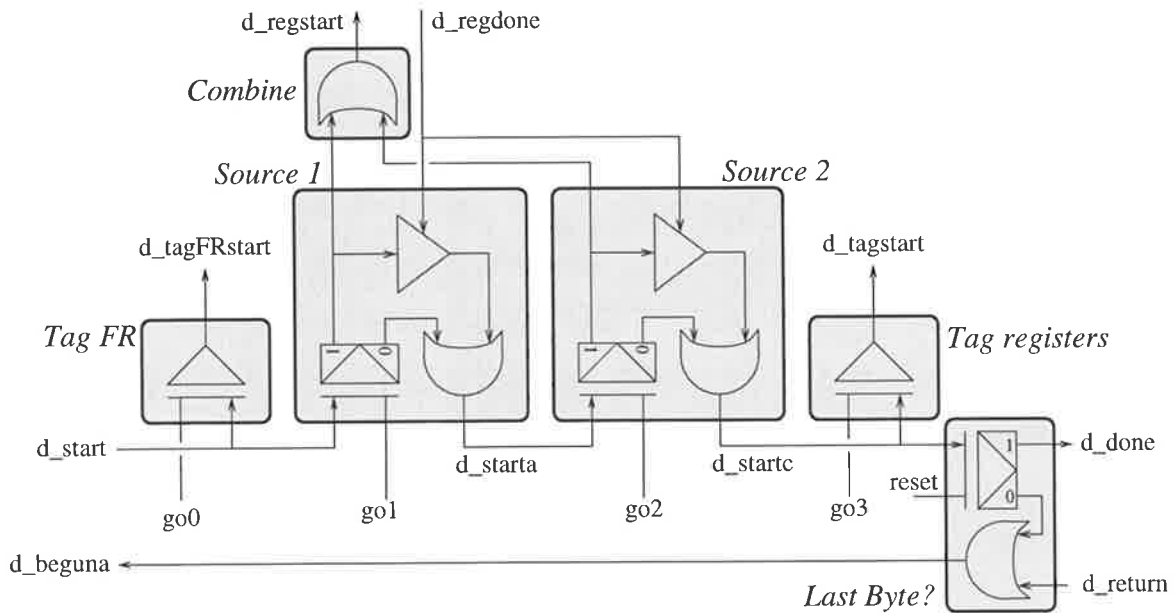


Figure 7.4: Event bypass method for controlling register accesses.

excluding the delay of the registers themselves. Although this may be the most intuitive implementation (which also happens to be a SI design excluding the tag operations), a better implementation based on conditional gating of logic signals, rather than events, is given in Fig.7.5

Essentially, the sourcing circuitry involves generating a logic level signal from an input event (via the *until* gate), which causes Δgo_{reg} to occur if the *go* signal is high. Otherwise, the register triggering is bypassed through the *done* signal by immediately passing out the event through the *send* gate. The signal *goreg* is used to trigger the register access operation, which therefore requires the registers themselves to be logic level triggered. This is in many ways advantageous, since bus precharging is easily accommodated without any additional control overhead, whereas for the strategy of Fig.7.4 the precharge signals would still have to be generated from the $\partial regstart$ event.

The register unit then must also supply a logic level completion signal (which is similarly advantageous), which then results in $\Delta done$ and sends out the initial input event. This *done* signal then resets the initially generated logic signal (from the *until* gate) which in turn resets the subsequent gates. The correlation between source and tag structures for this method and the previous is indicated by the shaded boxes. Note that an extra send gate is used here between source1 and source2 because a high signal on *regdone* sets both *done1* and *done2* high, so that were it not present the event $\partial startb$ would emerge prematurely as $\partial startc$.

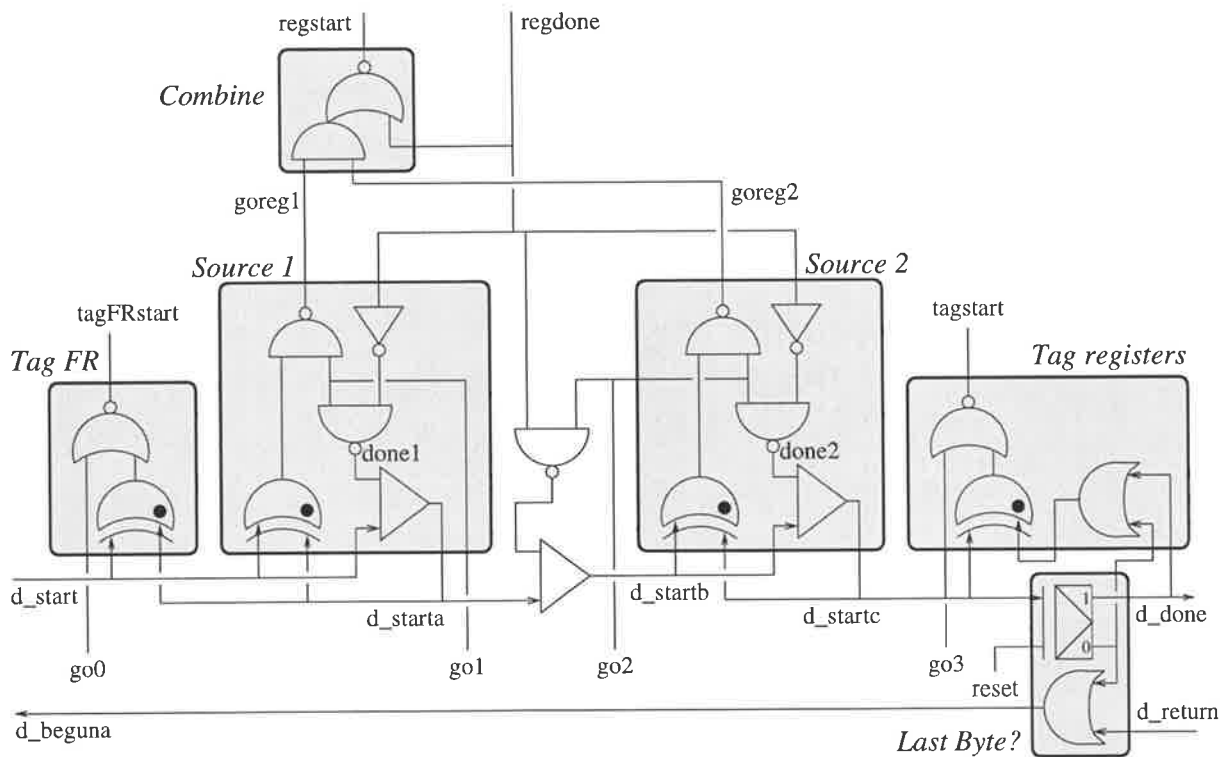


Figure 7.5: Logic bypass method for controlling register accesses.

This design results in a best case latency of $3t_{send} = 18$, a nominal latency of $3t_{send} + t_{xor} + 2t_{nand} + t_{andnor} + t_{inv} = 38$, and a worst case latency of $3(t_{send} + t_{xor} + 2t_{nand} + t_{andnor}) + t_{inv} = 74$. Comparing these results against those of the SI schema results in speed improvements of 55%, 41%, and 16% respectively (for which the first two scenarios are the most frequent), which are in fact further enhanced by the removal of a precharge generation phase from the event control. It should also be noted that the design area has been considerably reduced.

7.4.3 Adder, comparator, and stack processor (ACS)

The ACS is a 2 stage unit primarily used to compute the address offsets for LD and ST instructions using the register and offset modes, as well as computing PC relative branch locations. Since these operations involve 24 bit data signals it is necessary to implement a dedicated adder rather than re-use the 8 bit adder of the ALU (with 3 cycles), which would require greater complexity and hence slow down the processor speed.

The ACS stage also performs two other important tasks: branch detection and stack processing, in the first and second stages respectively. In the first stage, the contents of

the FR which were fetched from the preceding OF stage are compared to the branch code, and if a branch doesn't occur then the instruction is converted into a NOOP and the 24 bit adder is bypassed, otherwise the instruction proceeds as normal and all subsequent instructions are converted into NOOPs instead.

The second stage generates the SP addresses for the DC. A decrementing stack is used for which the top memory location can be set using the TRSP command. For a PUSH (or POP) operation the SP (or SP+1) address is supplied to the DC together with the fetched data from the OF stage, and then SP-1 (or SP+1) is stored as the new SP. A RETN instruction is more complex in that three POP operations are necessary to retrieve the new 24 bit PC location from memory, which can only store 8 bit data values. A CALL instruction is similarly problematic, requiring three PUSH operations to store the current PC location as well as having to reload the new PC address (giving a total of 4 operations for the one instruction). The following sections describe the operation and control schemas of the 24 bit adder, comparator, and stack processor in turn.

7.4.3.1 The 24 bit adder

The 24 bit adder is implemented in two stages each of which executes a 12 bit Manchester carry adder (MCA) [WE93, p322]. Although this requires a long chain of n transistors in the worst case, a large width:length ratio is used to reduce the computation time. Furthermore, since at best the execution rate of each stage will be for a two byte instruction (LD offset) the cycle time (and latency) of this adder is not critical, and merely needs to be less than twice the cycle time of the OF stage with one register fetch.

The first stage addition also pre-computes the output for the upper 12 bits assuming an input carry of zero. If this proves to be correct, then the second stage 12 bit adder does not need to be activated, thereby improving the system speed.

The control circuit for the first stage of the ACS unit is shown in Fig.7.6, and includes the control for managing both the adder and comparator sections of the design (shown in the shaded oval and square respectively). The state pipeline structure is used to implement the basic handshaking requirements between adjacent stages.

The control for the adder is simple. Once the pipeline begins its operation, as signalled by $\nabla acsse10$, the activation signal $act0$ for the MCA is set high provided that the instruction does in fact require the 24 bit adder (as given by $doadd$). A positive-edge only delay

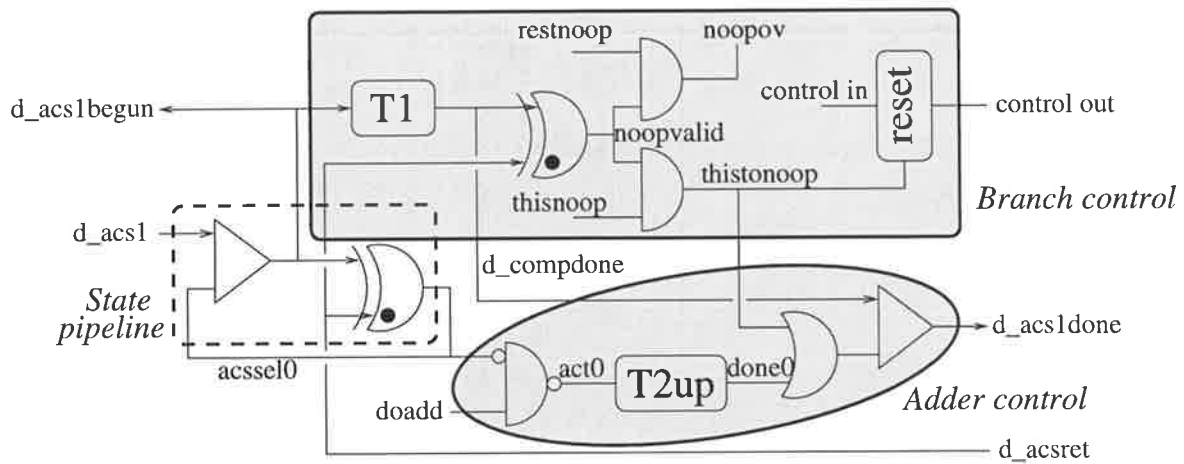


Figure 7.6: Control circuit for the first stage of the ACS.

unit is used to model the computation time of the 12 bit MCA, which is also partially absorbed into any subsequent control delays before the next stage's latching operation. Once the delay model signifies completion, the event $\partial compdone$ (delayed slightly from $\partial acs1begun$ to occur after the branch detection logic, as will be discussed next) is passed to the output event $\partial acs1done$. Note however that if a branch comparison indicates that it's not taken, then this instruction is converted into a NOOP (as signified by a high level on $thistonooop$), and $\partial acs1done$ is then passed out immediately since the result of the addition in this instance is irrelevant.

7.4.3.2 Comparator

The comparator is used to determine whether or not a branch is to be taken. This is implemented using a simple static logic tree which compares the flags from the FR against the branch code. Referring to Fig.7.6, if a branch is detected then $restnoop$ will go high, otherwise $thisnoop$ will go high instead. To prevent glitches on these signals from causing erroneous behaviour they are *and*'ed with the $noopvalid$ signal, which is itself generated from $\partial compdone$ and indicates when the branch detection logic is complete (using a simple delay model).

If $thistonooop$ then goes high (indicating that the branch isn't taken), the instruction is converted into a NOOP by bypassing the addition (as already mentioned) and masking the relevant control bits to indicate a NOOP to the next stage. If instead the $noopov$ signal goes high, then a branch has occurred, and all preceding stages are to be converted into NOOPs.

To implement this, the *noopov* signal is fed back to all preceding stages and sets a control signal *noopbit* high in every stage. The control state of each preceding stage's latches (opaque, transparent, or somewhere in between) is irrelevant, since the propagation of this signal through a stage is less than the pulse width of *noopov*. There are only two places where a potential problem exists: in halting the PC until its new location has been loaded, and in the tagging operation of the preceding OF stage.

The first problem involves a metastability issue in the PC unit, which is discussed in Section 7.4.7.2, and the second involves ensuring that a tag operation for the next instruction is not initiated before it has been converted into a NOOP. A register fetch would be okay, since its results could be discarded, however undoing a tag operation would be a complex procedure. This problem is in fact solved without effort, since the time from $\partialacs1begun$ to $\Delta noopov$ is less than the the return time to handshake with the OF stage and initiate a tag. If it were not, then the tag operations in the OF stage would merely have to be delayed slightly, an issue which again does not influence the critical control path of the OF unit.

7.4.3.3 Stack processing - single operations

The stack pointer is a 24 bit address indicating the current location of the top of the stack, and is multiplexed onto the address bus at the ACS output for a stack operation. This can be set with a TRSP instruction, which loads the SP with the contents of Q12.

The stack is implemented with an incrementer and a decrementer. For a PUSH operation the current SP value is required, whereas a POP operation requires SP+1. Therefore for a POP operation SP+1 is loaded into the stack *prior* to the next stage being activated, and for a PUSH operation SP-1 is loaded in *after* the next stage is activated.

Figure 7.7 shows the control circuit used for this operation, together with the state pipeline controllers for the handshaking with adjacent stages and the control for the upper 12 bit adder (which is essentially the same as that already described, except that in this instance *doaddhi* is low if the pre-computation for $c12 = 0$ in the previous stage was correct). A pulse is generated for when the stage is first activated (from $\partialacs1ret$ through the *merge* and *send* gates) and also after the next stage has been activated (from $\partialacs2done$ through a *send* gate). A POP operation selects the first of these to reload the SP (provided that a SP operation is occurring), and a PUSH selects the second.

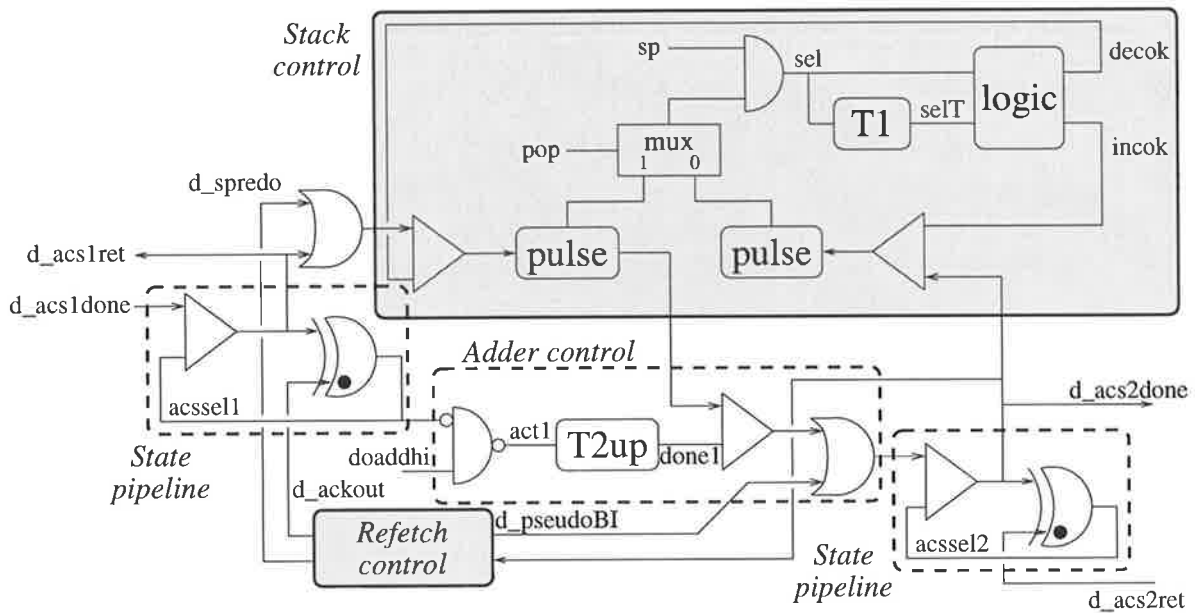


Figure 7.7: Control circuit for the second stage of the ACS.

Once loaded, the increment and decrement of the SP are initiated (from the *sel* pulse). These units are not self-timed and use a simple static logic chain for their computation, which requires the worst case propagation time to be managed. This is implemented with a delay on the pulse signal to indicate completion (which requires careful management to ensure that the negative edge propagation is no faster than the positive edge) from which the *incok* and *decok* signals are produced. These go high when SP+1 and SP-1 are valid, and prevent their re-loading into the SP from a subsequent operation until such time.

The CALL and RETN instructions require 3 stack operations to store or retrieve the PC, which complicates the control schema. A refetch controller is needed to repeat the SP operation 2 more times (via $\partial sprede$), and for the CALL instruction an additional operation is required to load the PC with its new location ($\partial pseudoBI$). These signals are incorporated into the single operation control schema with the two *merge* gates, and are generated as follows.

7.4.3.4 Stack processing - refetch control

The control circuitry for implementing the CALL and RETN instructions is shown in Fig.7.8. Once the following stage has been initiated $\partial acs2done$ occurs, which then propagates through a *select* gate for which *sprefetch* is initially high for a CALL or RETN instruction (as given by *pcwrite*), and which stays high until the second event on $\partial sprede$

occurs. Assuming $call2$ is low (for a RETN instruction), this will cause 2 events to occur on $\partial spreado$ before $\partial ackout$ occurs to indicate completion to stage 1, giving a total of 3 SP operations as required.

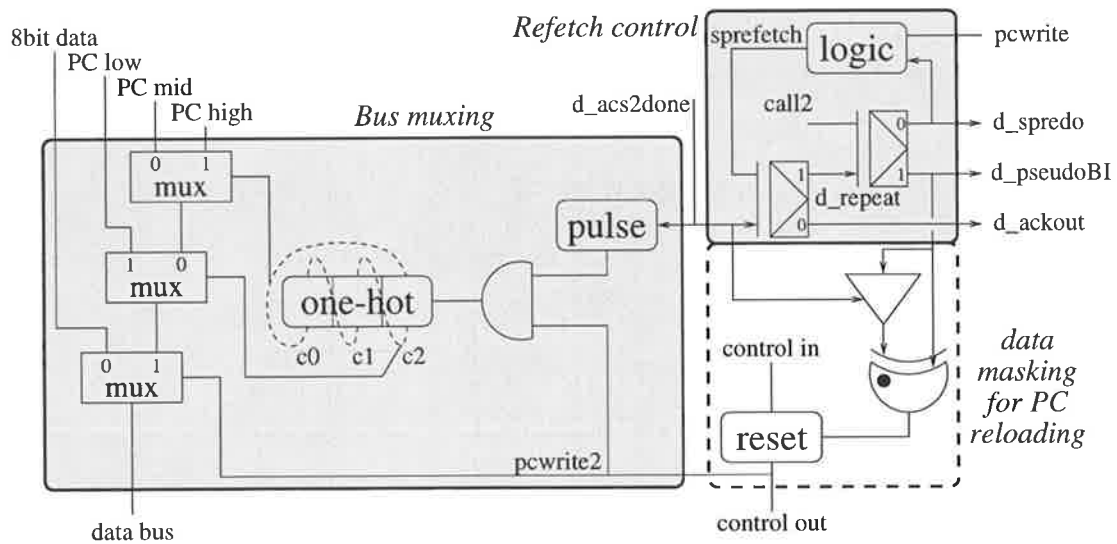


Figure 7.8: Refetch control for the second stage of the ACS.

If however a CALL instruction is occurring, then the first event on $\partial repeat$ will be transferred to $\partial pseudoBI$ ($call2$ is high for this first event only, and low thereafter). This masks the control signals to initiate a reload of the new PC location until $\partial acs2done$ occurs (once the control signals have been latched). For the next iteration $call2 = 0$ and the control masking is removed, so that as per the RETN instruction 2 more SP operations are initiated before $\partial ackout$.

The purpose of these iterations is to enable the PC to be loaded onto the stack in three 8 bit segments. Each time $\partial acs2done$ occurs a one-hot counter is incremented which selects the appropriate segment onto the data bus. The high byte is stored first and the low byte last, so that when a RETN instruction reloads these into the PC the incrementing can occur immediately on the low byte (since PC+1 is actually required as the RETN target).

7.4.4 Arithmetic and logical unit

The ALU performs addition operations (including subtraction, negation, incrementing, and decrementing), logical functions, and shifting of 8 bit operands, as well as passing out the input data directly to the output for a move instruction. An 8 bit MCA is used

to implement the additions, and since the ALU is at best triggered at a rate equal to two cycles of the OF stage with one register fetch, the latency of this unit is not critical.

The ALU is structured such that the logical, shifting, and move operations are all performed in parallel, and the dynamic adder is only triggered (also in parallel) when necessary. The appropriate result is then multiplexed onto the output bus and the relevant flags computed (any flags which are not of relevance for a given operation remain unchanged), which are then written back to the register bank.

The control circuitry for the ALU is essentially the same as the adder control of Fig.7.6, with a constant delay element in the event path (equivalent to $T1$) to model the computation time of the logical and shifting operations, and a positive edge delay used to model the computation time of the adder. If the adder is not used, then the delayed event is passed out directly through the *send* gate (effectively replacing *thistonoop* with \overline{add}).

7.4.5 Order unit

The ACS stage can issue an instruction to either the ALU or the DC (and the PC too for a branch), which after processing may then wish to write back a result to the register bank. Since these units can initiate a write back at any time there is a contention issue to be resolved, for which there are two possible solutions: using an in-order execution model ensures that the units write back their results in the same order as the original instruction stream; whereas an out-of-order execution model allows these to occur at any time, provided only that writes to the *same* register remain in-order.

Although the latter model prevents unnecessary stalls for when one unit has to wait for a slower, but earlier, computation to finish in another unit (as may happen in *ECSTAC* if a miss in the DC causes a subsequent ALU operation to stall), the control strategy for initiating a write back and especially for maintaining the register scoreboard is more complex. In *ECSTAC* the frequency of out-of-order write backs is expected to be low, therefore a simple in-order execution model is used, and the OU is employed to effect this.

When a DC or ALU operation is activated from the ACS stage (which will eventually initiate a write back), a flag is also loaded into the OU which records which unit was used. The OU is essentially a FIFO and maintains the instruction order from the ACS. When

the ALU or DC then initiate their register write, the output of the OU is first checked to ensure that the correct execution order is being achieved. If a latter ALU instruction completes before an earlier DC instruction it will stall until the DC has completed.

To implement such a design it is necessary to have a FIFO length at least as long as the maximum number of stages which can be filled after the ACS, otherwise the OU may unnecessarily create a stall at the ACS output (and must then also issue a return acknowledgement). By considering Fig.7.1 there are at most 5 stages which can be filled before the ACS stage itself stalls, therefore the OU is designed with a 7 stage FIFO.

The general structure of the OU is shown in Fig.7.9. This design enables the latency of the FIFO to be very small and independent of the FIFO length, which is advantageous since an ALU operation can potentially occur very quickly, and may therefore stall unnecessarily before writing back if the FIFO latency was longer. Note that the OU also multiplexes the appropriate data bus to the register unit as well as generating the return events for the ALU and DC from the register bank.

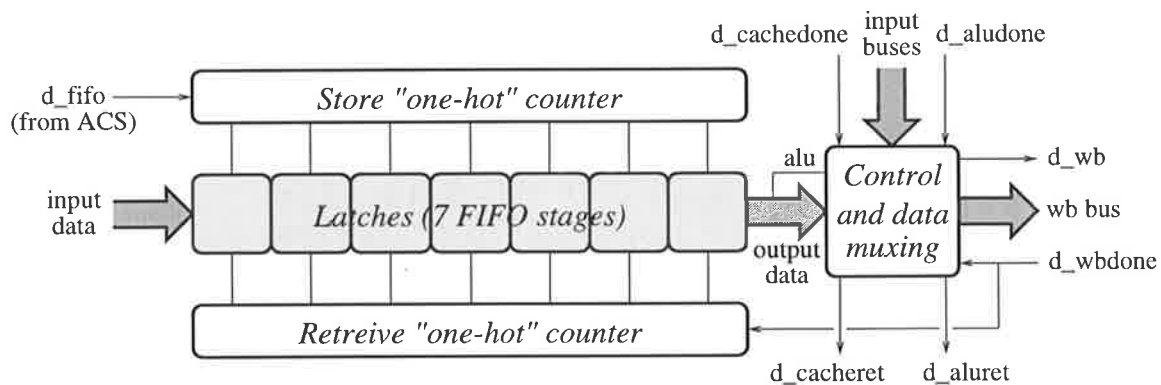


Figure 7.9: General structure of the low-latency FIFO used in the order unit.

7.4.6 Registers and scoreboarding

The register bank (designed by Sam Appleton) is simply a 16x8 bit array of register cells employing a standard two read port, one write port design. Note that although only two read ports exist, a Qfetch (fetching 3 register values instead of just one) can still occur since the extra two fetches always go to a unique output bus (bus2 or bus3 only). A separate register is also implemented for the flags, which reads onto bus1 (as does a one byte Rfetch) and can be written to at the same time as a write back to a data register.

For any microprocessor it is essential to prevent the reading of a register when new data values are still to be written, termed a “read after write” (RAW) hazard. Conveniently, a “write after read” (WAR) hazard is handled implicitly by the pipeline structure as is a “write after write” (WAW) hazard in conjunction with the OU, however RAW hazards require specific attention. One possible method of handling these is to have two $\log_2 n$ bit counters per register (where n is the number of pipeline stages between reading and writing) which continually increment whenever a tag set or reset occurs. When these two counter values are equal the register contents may be sourced, however this approach is significantly area intensive. Another solution is presented in [PDF+92] in which a FIFO of width m (where m is the number of registers used) and length n is used to store a bit for each register indicating if it needs to be written to. These are then *or*'ed across each FIFO stage to determine the readability of each register. This approach is still area intensive and scales linearly with the number of registers *and* the number of pipeline stages.

ECSTAC employs a very different method which uses a row of latches at least as numerous as the number of potential pipeline stages, and stores the actual encoded *destination register* itself (of width $\log_2 m$) as shown in Fig.7.10. Provided that the ordering of instructions is maintained (an issue demanded by this strategy), then the hold and pass states of any one latch may be regulated by the tag’s start and reset signals in a simple “one hot” fashion. The number of *cells* is simply replicated for at least as many pipeline stages as necessary.

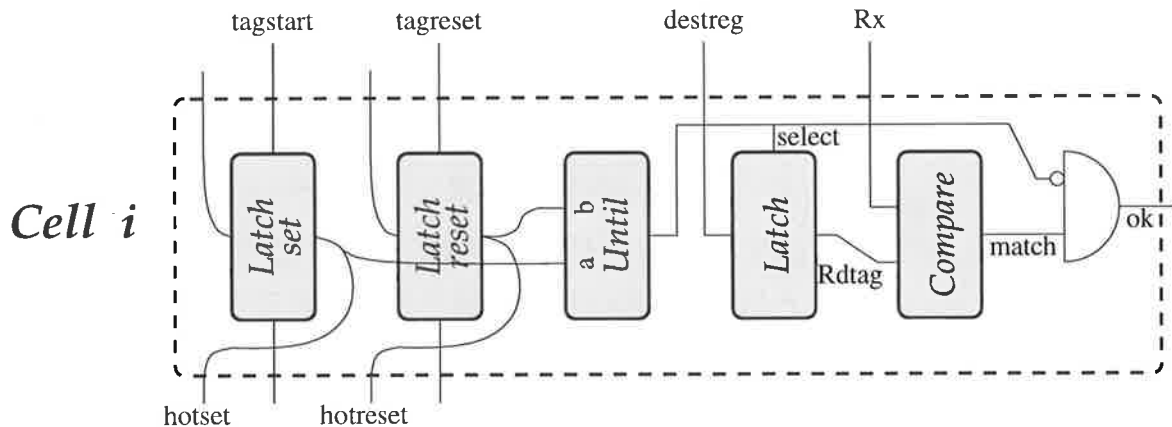


Figure 7.10: A tag cell used in the register scoreboard.

When a request to read register Rx occurs, it is compared against the previously latched destination register in each cell and sets ok_i low if they match. All of the ok_i

signals in the array are then *and*'ed together to give the readability of the source register Rx (in practice, a single complex gate is used to perform the match and OR operation), and if low then the read cycle into the register bank is halted. When a write back occurs, the earliest cell's *select* signal will go high invalidating the previously latched destination register (which has just been written to), and subsequently setting ok_i high to activate the register read with valid data (unless another cell indicates that yet another write back to this register is still to occur). Note that in practice there are actually three comparators with inputs $Rdtag$, $Rdtag - 1$, and $Rdtag - 2$ respectively, which are used to determine the readability for the three source registers of a Qfetch.

This scoreboarding (SCB) scheme requires significantly less area than the two methods mentioned above and yet still enables the rapid detection of the readability of a particular register. Furthermore, this implementation scales logarithmically with the number of registers used.

7.4.7 Program counter

The PC unit interacts with three separate entities: the DC and ACS outputs for writing to the PC (for RETN and BRANCH instructions respectively); and the IC for reading the next PC location. Therefore the PC structure is broken down into two sections as shown in Fig.7.11, with the first being used to distinguish between a PC write from the Cache unit or a branch instruction (*Write Interface*), and the second being used to distinguish between a PC read or write operation (*PC Controller*).

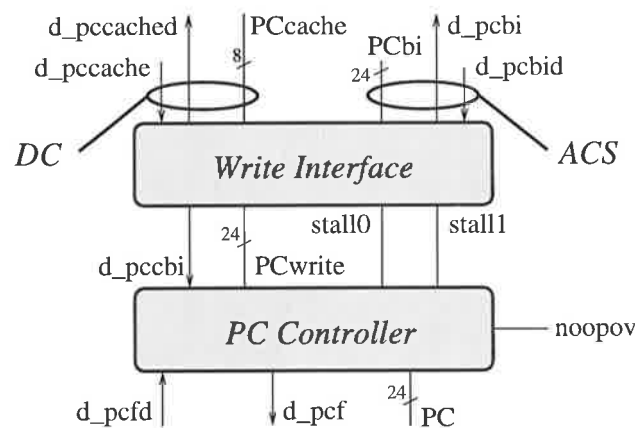


Figure 7.11: General architecture of the PC unit.

7.4.7.1 Write interface

This unit receives request events from the ACS and DC units together with the associated data which is to be written to the PC, and then initiates this write cycle and upon completion generates the return event to whichever unit called it. Note that it is impossible for coincident input events from these two units to occur, because as soon as a PC write is detected in the first ACS stage, all subsequent instructions (in preceding stages of the pipeline) are then converted into NOOPs and the PC fetching cycle (from the IC) is halted until the new PC value has been written. The write interface circuitry is shown in Fig.7.12.

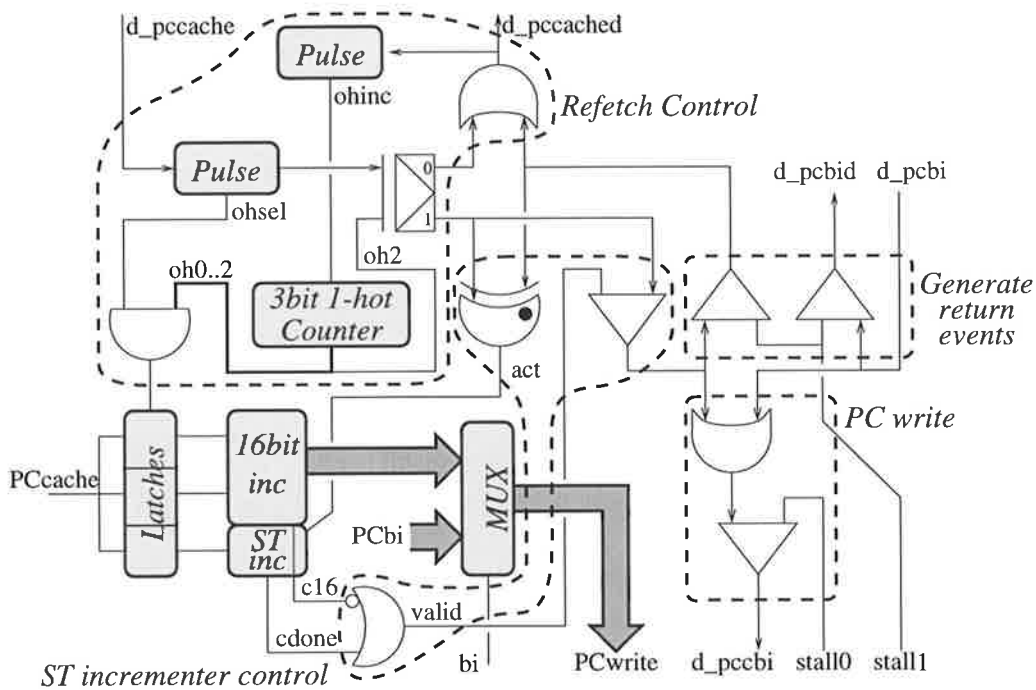


Figure 7.12: Interface circuitry for the DC and ACS write back phases to the PC.

The data which comes from the ACS output is the new 24 bit location of the branch instruction ($PCbi$), but the data which comes from the DC ($PCcache$) is in three streams of 8 bits, each of which represents the new PC value from low byte to high byte respectively. Therefore additional refetch control must be implemented for the DC events which enables the full 24 bit PC address to be latched before sending out the write event to the PC unit itself. This can be done in similar vein to the SP refetch control used for CALL and RETN instructions in Section 7.4.3.4, using a “one-hot” counter to multiplex the incoming byte into the appropriate portion of the 24 bit address.

An increment of this address must still be performed. This is implemented with a simple static logic incremter for the low 16 bits, since an 8 bit increment can safely occur before the next byte is loaded from the DC. If the increment isn't yet completed when the last byte is loaded (implying $c_{16} = 1$, and occurring only 1 in 2^{16} times), then the last 8 bits are incremented in a self-timed fashion.

Once the new PC location to be written is available, the PC write operation is activated via $\partial pccbi$. The signal $stall0$ is used to halt this write phase until the instruction fetch cycle has been stalled, and $stall1$ is used to halt the acknowledge events until the new PC address has been loaded.

7.4.7.2 PC controller

When a branch occurs the signal $noopov$ pulses high, which causes $hpfy$ in Fig.7.13 to go high and therefore halt the PC-IC instruction fetch cycle according to the metastability resolving technique of Section 4.4.1 (as indicated in the figure).

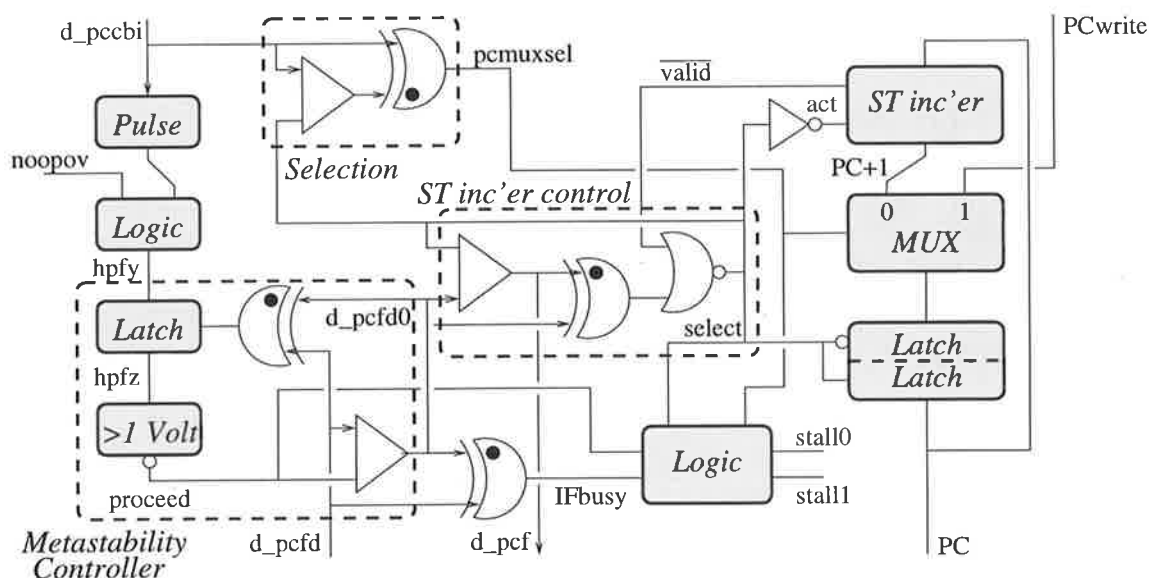


Figure 7.13: Control schema for the PC unit.

Once the new PC location is ready to be loaded, $\partial pccbi$ occurs and generates a write pulse to the *logic* gate which then sets $hpfy$ low. This event also sets $pccmuxsel$ high to multiplex in the new PC location ($PCwrite$), as opposed to $PC + 1$ as would otherwise be used in a typical fetch cycle. The signal $hpfy$ will eventually propagate through to the *send* gate and produce $\partial pcf d0$, which then latches in the new PC location and then

restarts the PC-IC instruction fetch cycle.

If however there was no program branch and therefore no *noopov* pulse, then the input event *∂pcfd* will simply propagate through to latch in PC+1 for the next fetch cycle. Whilst the IC is fetching new data, the next incremented PC location is generated using a self-timed incrementer, and the \overline{valid} signal is used to prevent the latching of the new value until this incrementing has completed.

7.5 Testability issues

In an asynchronous system each section of the design has to have an explicit control schema designed for it, whereas a synchronous system is globally governed by the clock. This increase in control complexity means that there is a greater chance of design failure, either within each section of the design or in the interfacing between them. In addition to this there is still the same possibility of a failure in the data path logic (say, for stuck-at faults) which could also result in the design not functioning to specification.

If after fabrication it is found that the chip does not function correctly, it is of course necessary to determine the cause of the problem(s). This might then enable corrective measures to be applied to the chip to give at least some degree of functionality, or at worst should enable the problem to be rectified in a subsequent design iteration.

The *ECSTAC* processor employs three primary methods for analyzing and repairing circuit faults, which were developed in co-operation with Sam Appleton.

7.5.1 Delay modelled *Vtt* bus

It is sometimes the case that a chip may be fabricated with transistor characteristics which differ significantly from those that were used in the simulations. If this is the case then it's possible for the computations in the data path to take longer than anticipated, or alternatively the control path may be faster. In a synchronous system a slower process corner can be overcome by reducing the clock frequency so that the data has sufficient time to compute, however since an asynchronous system is locally controlled this same principle cannot be employed.

Instead, the *ECSTAC* processor utilizes a separate *Vtt* power bus which drives all of the delay modelled elements in the design (such as *T1* and *T2up* of Fig.7.7). If the control

path operates faster than the data path due to a hazardous process corner, then the delay elements which are used to model this can be increased by lowering the supply voltage of V_{tt} . As such it may then be possible for an erroneous chip to function properly.

Note that it is not possible to rectify any problems which arise from non-acknowledged control structures which were deemed to have a shorter latency than cycle time, however the likelihood of this problem is expected to be low. Although not implemented in *ECSTAC*, a prudent solution could be to use a separate power bus for the cyclic and non-acknowledged control paths, and slow down the former (reduce its supply voltage) if the non-acknowledged paths are sufficiently slow enough to cause a problem.

7.5.2 Interface delays

During the design process every major functional blocks of Fig.7.1 was simulated in detail using Hspice, which enabled accurate timing information to be attained. However, since the complete design required too much memory and processing time to properly simulate in Hspice, an event-driven simulator IRSIM was used to test the entire system.

Although this simulator uses a parameter file which closely matches the Hspice simulations of smaller systems, it is still less reliable. As such it is possible that although each functional block is simulated as confidently as possible, there is less confidence in the accuracy of the interfacing between blocks. It is conceivable that assumptions regarding the timing of the input data and events for any one block may in fact be violated by the block preceding it.

To overcome this potential problem, a simple selection unit was placed in the forward event paths between each sub-system which enabled a lumped delay to be switched in prior to start-up (when the initialization signal *init* goes low). Therefore if the input event to a unit occurred before the data had been fully computed, then inserting this delay may solve the problem and enhance the chip's functionality.

7.5.3 Scan testing

Computational errors in the data path (as opposed to control errors in the event path as have thus far been addressed) can be identified using scan test registers, of which one such register cell used in *ECSTAC* is shown in Fig.7.14.

One of two data signals (which may include the data level of an event line, for testing)

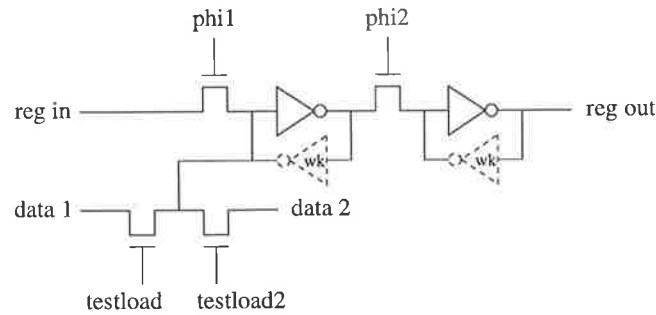


Figure 7.14: Register cell used for scan testing the outputs from each stage.

is first loaded into the scan register by pulsing *testload* or *testload2* high. This loads parallel data into the n register cells used in the scan chain, which can then be read out serially by activating the two phase non-overlapping clock signals (which must both be off when data is first loaded in). Scan registers are placed at the outputs of each sub-system to determine whether or not the control and data signals have computed correctly, therefore isolating the source of any problems to the block level. This can then be pin-pointed within the block via simulation and rectified in a subsequent iteration of the design.

Note that the data must be valid when loaded into the registers, however since an asynchronous control schema is used it is not possible to know exactly when this is so. Therefore it is necessary to have each stage's output stall when scanning is required, and to pulse each unit's output event (if pending) into the following unit once the data for each instruction has been scanned. Since this pulse width (to a *send* gate) ought to be small to prevent the possibility of more than one cycle occurring, it is generated on-chip from an external input event. If this event is initially low, then the chip is running "at speed" with no scanning (and therefore no stalling between stages), otherwise each subsequent input event after start-up will generate an on-chip pulse. In many ways, the block level hierarchy of the system then acts like a clocked design, with each unit processing one cycle per external event, if required. Clearly, the events must be sufficiently spaced to allow the worst case processing of a stage, and the loading and subsequent scanning of its data.

7.6 Simulation results

The control structures presented in the preceding sections were first simulated in VHDL with code produced from a tool which converted from a temporal specification of the

circuit. Once these were deemed functional they were then custom implemented in the ES2 technology using the mask layout editor MAGIC.

For each sub-system the data path was first implemented and simulated in Hspice. Then the control schema was floorplanned and laid out to match the height of the data path and to produce the required control signals (such as for multiplexing, masking, and latching) as near as possible to their location in the data path. The control layout was then simulated with capacitive loads for control signals taken from the data path simulations, and then finally the entire block was simulated with instruction traces designed to test the full functionality required of the unit.

7.6.1 Sub-system simulations

Table 7.2 shows the statistical information and performance characteristics of each sub-system of the processor, with all times quoted incorporating the handshaking delay of adjacent state pipeline stages (if relevant). The total area quoted is for the final design prior to fabrication, and the total number of transistors includes those used for testing. The cycle time (σ), latency (λ), and power consumption (P) were averaged over numerous random instruction traces.

Block	Size(mm)	Area (mm ²)	Transistors	σ_{ave} (ns)	λ_{ave} (ns)	P_{ave} (mW)
ID	0.67*0.85	0.57	4722	10	8	50
OF	0.46*0.63	0.29	2058	12	20	23
ACS	0.75*0.53	0.40	4518	25	17	37
ALU	0.40*0.31	0.12	1307	9	7	38
REG	0.61*0.39	0.24	1866	7	4	56
SCB	0.43*0.40	0.17	2167	4	2	13
OU	0.15*0.62	0.09	1188	8	5	12
PC	0.34*0.44	0.15	1954	8	6	17
Total	2.30*1.64	3.78	21093	-	-	-

Table 7.2: Statistical information from Hspice simulations of each *ECSTAC* sub-system.

Note that the OF and ACS stages exhibit significant variations in cycle time and latency due to the possibility of multiple register fetches and stack operations respectively. The PC can also exhibit a longer latency if the carry chain is long, however this is a rare occurrence.

One may surmise from this table that the cycle times of the ACS and OF stages

(when combined with register accesses) will be the dominant factor regarding the overall processing speed. The average cycle time of the ACS stage is 25ns, and it has been determined that when the register and scoreboarding delays are incorporated into the OF stage, an average cycle time of approximately 16ns results. Since there is an average of approximately 2.2 bytes per instruction the total cycle time of the OF-SCB-REG cycle will dominate. A performance of approximately 28 Mips may therefore be anticipated, however this will be degraded by the back propagation of a longer latency operation in either the ACS or OF stages, as well as the delay involved in determining and resolving a branch instruction. These effects can be quantified by a detailed simulation of the processor core.

7.6.2 Core simulation environments

The processor core was simulated in IRSIM but without either of the caches in place. This enabled a shorter processing time as well as providing information on the speed of the section of the design implemented by the author. The DC was replaced by a circuit which returned the required data (or stored it) with an approximate cycle time of 10ns, and is comparable to the actual cycle time of the DC unit [AML95a]. This is effectively equivalent to assuming a DC with a 100% hit rate.

Before implementing an “at-speed” test of the core, it was necessary to determine the functionality of the entire system for each instruction. To effect this, the data and PC values into the ID stage (which would normally have come from the IC) were encoded directly for each instruction, with a long cycle time between successive bytes to ensure that each one had sufficient processing time in the worst case. Although this didn’t fully test the scoreboarding or block interfacing control, it still enabled many aspects of the overall architecture to be tested and fine tuned.

Once functional at this level, the IC was replaced by a FIFO in which each byte of an instruction stream was stored. At start-up, the first FIFO value would be read out and passed into the ID together with the PC location direct from the PC unit. The output of the ID stage then activated the PC unit and grabbed the next value from the FIFO, and so on. This enabled the core to be tested at its expected operating speed, and is equivalent to using an IC with a 100% hit rate.

7.6.3 General purpose instruction streams

Due to processing time and memory limitations, the FIFO was restricted to 450 bytes in length, which equates to around 200 instructions in a typical stream. This prevents the core from being tested with reasonably complex benchmark programs to determine its speed, therefore various sets of random instruction traces were used to gauge this with relative instruction frequencies as given in Table 7.3. This distribution is based on the dynamic instruction traces given in [HP90, Chapter 4]. Note that for every CALL instruction a corresponding RETN is also eventually issued.

Special		Stack		ALU		Branch		Memory	
Inst	Freq	Inst	Freq	Inst	Freq	Inst	Freq	Inst	Freq
NOOP	0.4	POPF	2.0	short reg1	30.0	CALL	3.0	LD reg	4.1
FLSH	0.3	PSHF	2.0	short reg2	8.0	JUMP off	2.0	LD off	0.8
ICDS	0.3	POP	2.0	short off2	2.0	JUMP reg	0.5	LD dir	2.6
ICEN	0.4	PUSH	2.0	long reg2	7.0	JMPc off	7.5	ST reg	4.1
DCDS	0.3			long off2	13.0	JMPc reg	2.0	ST off	0.8
DCEN	0.3							ST dir	2.6
total	2.0	total	8.0	total	60.0	total	15.0	total	15.0

Table 7.3: Relative instruction frequencies used for the general testing of *ECSTAC*.

Preceding any generated instruction stream are 5 instructions (12 bytes) which are used to initialize the SP. The first four are ALU operations which set the relevant bits in Q12, and the fifth executes a TRSP instruction.

A number of instruction streams were generated which used either the ALU, the DC, or both, and with branch instructions either included or excluded from the trace. This enabled the effects of branch delays to be measured as well as determining the relative operating speeds for ALU and memory operations. The results of this simulation are given in Table 7.4, and all speeds are averaged over numerous instruction traces of the same type to enhance the accuracy.

Stream Type	Mbps	Mips
All + branch	29.1	13.4
All - branch	33.4	14.8
Cache only	32.6	15.8
ALU only	34.0	14.5

Table 7.4: Simulation speeds of *ECSTAC* for unit specific instruction streams, quoting the number of bytes (Mbps) and instructions (Mips) processed per second (in millions).

Since there is a variable number of bytes per instruction the Mbps field (millions of bytes per second) is also quoted. This figure gives an estimation of what the processor speed would be were all instructions able to be encoded in one byte with equal address and data widths. The Mips speed for “All - branch” is seen to be approximately half of what was anticipated in Section 7.6.1, which is due to the back propagating effect of slower operations in the OF and ACS stages, so that the overall cycle time of the processor is in fact closer to the worst case than the average. This verifies that asynchronous pipelining does not enable the average case processing time of each stage to be properly utilized.

It can be seen that the effect of branching operations is to reduce the Mips and Mbps by about 10%. This is because the delay from fetching the branch instruction to branch resolution involves propagating all the way through the processor core, and any instructions fetched in its shadow are irrelevant. By placing not just the detection of a branch but the updating of the PC as well at the start of the pipeline, or perhaps even decoupling the process from the pipeline completely, the penalty for program branches could be reduced.

It is interesting to note that the processing speeds for the ALU and cache operations show very little variation. This implies that any bottlenecks are not *unit* dependent, but are caused by other factors not evident from these simulations.

ALU operations are in fact slightly faster than cache operations in terms of Mbps. This is because the ALU has a lower latency through the ACS stage than do cache operations, which may require stack processing and 24 bit additions for address locations. However, since ALU operations typically require more bytes in their encoding (2.4 bytes versus 2.0), the Mips rate is slower than for cache operations.

It should also be noted that the effect of register hazards is minimal on the processing speed. This is because of the low latency of an ALU operation to write back its result from the OF, so that in general only the immediately following data dependent operation can get stalled. Furthermore, it is common for the following instruction to source the data dependent register on its second byte, by which time the result has often been written back.

Given the typical speed of the processor for the most general instruction stream, it is possible to estimate the power consumption of the processor (since this is not provided directly by the simulator). By averaging the power dissipation for each unit (from Table

7.2) over the typical cycle time of the processor, and factoring in the frequency of usage of each unit after the ACS from Table 7.3, an estimate of the total power consumption can be made. These estimates for each unit are given in Table 7.5, from which the overall power dissipation of the processor for a typical instruction trace is estimated to be 58 mW.

Unit	ID	OF	ACS	ALU	REG	SCB	OU	PC	Total
Power (mW)	14.5	10.6	12.4	2.8	11.4	1.5	0.9	4.0	58.1

Table 7.5: Power estimations of each unit for a typical instruction stream.

7.6.4 Instruction streams for determining bottlenecks

In an effort to locate any bottlenecks in the system, a series of instruction traces were generated to test those sections of the processor that were deemed to be potentially problematic. In particular, the following three factors were investigated: the number of register sources per instruction (to test the effect of operand fetching); the number of bytes per instruction (to test the effect of variable byte instruction encoding enforced by the data and address mismatch); and the number of stack processing operations required (to test the effect of the refetch control, again a byproduct of the mismatch between buses). The results of these instruction traces are shown in Table 7.6.

Stream Type	Mbps	Mips
No reg sources	43.1 ²	22.4
1 reg sources	33.9	15.7
>1 reg sources	27.7 ¹	12.1 ³
1 byte insts	28.5	28.0 ⁴
2 byte insts	31.2	15.5
>2 byte insts	36.7 ²	11.5 ³
SP insts	27.5 ¹	27.0 ⁴
No SP insts	33.7	14.3

Table 7.6: Simulation speeds of *ECSTAC* for *bottlenecked* instruction streams.

It is evident now that significant variations have occurred in both Mbps and Mips, thereby enabling the causes of processor bottlenecks to be identified. By first considering the Mbps, a low value results (note 1 in Table 7.6) when multiple register or SP fetches are required for the one instruction. A high Mbps (note 2) results when the instruction does not have to access the registers, or for long instructions in which bytes 3 or 4 typically

do *no work*, and pass through the pipeline directly. This implies that to achieve a high Mbps it is necessary to remove refetching operations.

In considering the Mips a low value results (note 3) when a large number of bytes are required per instruction (which also covers the case of >1 register sources), and conversely a high value results (note 4) when only one byte is needed per instruction. Note that in this instance the Mbps \neq Mips exactly (as would be expected) because of the 5 instructions (12 bytes) needed to initialize the SP.

Removing the refetching operations and encoding all instructions in the same single byte length will therefore remove the bottlenecks in the processor and improve its speed. As could be expected, these control requirements are a direct result of the mismatch between data and address widths. Therefore if these were identical (say, 32 bits each) then a significant Mips improvement would result.

Although the Mbps gives an indication of this improvement, it does not incorporate the fact that the control structures would be simpler for a matched 32 bit machine, and therefore the processing speeds of each unit for the typical case would be even faster. As a rough estimate, the processor speed for such a machine (*ECSTAC-32*) would probably approach 50 Mips for a typical operation (approximately 1.7 times the Mbps of Table 7.4, and equivalent to a 40% average reduction in control cycle times).

The *ECSTAC* processor has been fabricated using the ES2 technology, and a microphotograph of the chip is shown in Fig.7.15 with each section of the processor core appropriately labelled at the top of the figure. Of course prior to fabrication, the core of the processor was integrated with the DC and IC structures and tested to ensure proper functionality between them. This chip is currently being evaluated.

7.6.5 Comparisons

Table 7.7 provides some of the important performance characteristics of the *ECSTAC-32* processor together with those of other asynchronous CMOS microprocessors previously reported. To give an estimate of the performance of a 32 bit implementation of *ECSTAC*, the size and power values given in Tables 7.2 and 7.5 respectively have been scaled by $32/24 \approx 1.3$ (except for the ALU which has been scaled by $32/8 = 4$), and the power dissipation has also been scaled (as relevant) by the increase in operating speed surmised

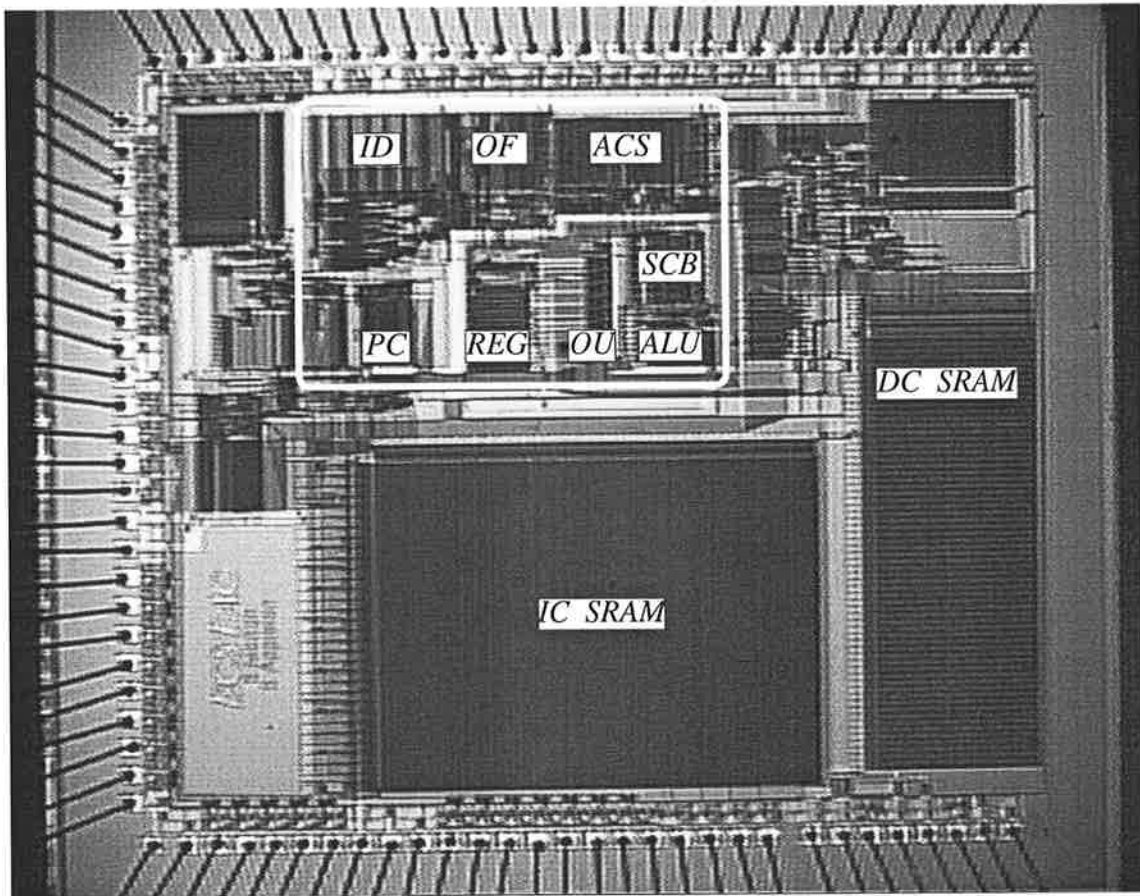


Figure 7.15: A microphotograph of the *ECSTAC* microprocessor.

in Section 7.6.4 for the architecture with a matched data and address width. Note that scaling of the other processors into the same technology and data width as *ECSTAC-32* will not enable a more accurate comparison between them because their ISAs are so vastly different.

The transistor count and area of *ECSTAC-32* is the lowest of the 32 bit processors, and would probably still be lower than the TITAC-I [Nan95] and Caltech [MBL⁺89b] processors if implemented with a 32 bit data path. The transistor density of *ECSTAC-32* is less than AMULET2 [MU] but still greater than the other processors. Although there are many factors involved here (such as technology and architectural differences), this data still supports the hypothesis that the ECS approach enables low area circuits to be devised.

ECSTAC-32 also gives the best speed performance and is significantly better than any of the 2P paradigms, but this improvement is reduced when compared to a 4P paradigm. This gives further weight to the notion that although 4P is better than 2P in delay

Processor	Caltech	TITAC-I	Amulet1	Amulet2	ECSTAC-32
Design style	4P QDI	2P QDI	2P SI	4P SI	2P BD
Bit width	16	8	32	32	8 & 24
Technology (μm)	1.6	1.0	1.0	0.5	0.7
Vdd (Volts)	5	5	5	3.3	5
Area (mm^2)	19.3	??	22.6	12.5	5.9
Transistors (k)	≈ 20	≈ 22	58.4	93	26
Density (T/mm^2)	1.0	??	2.4	7.4	4.4
Speed (Mips)	18.0	11.2	11.7	42	50
Power (mW)	225	212	152	150	90
Mips/Watt	80	53	77	267	556

Table 7.7: Comparison of performance characteristics of various asynchronous microprocessors.

insensitive and speed independent environments, it is not necessarily better in a bounded delay model. In fact, the 2P BD model used in the ECS methodology is shown to be superior to the 4P models used elsewhere. To the credit of the AMULET group however, they have implemented a commercial ISA (of the ARM microprocessor) whereas all of the other processors use their own ISA which obviously enables a greater degree of flexibility in their implementation.

The AMULET processors were designed for low power operation, however the *ECSTAC-32* processor was designed for high speed with the expectation that low power dissipation would still result without explicitly designing for it. The power dissipation and Mips/Watt figures in Table 7.7 also support this hypothesis.

7.7 Summary

The reason for implementing an asynchronous microprocessor using ECS techniques was to determine the speed advantage which could be achieved when applied to a complex design, and to see whether or not low power dissipation still results. Although as implemented *ECSTAC* is only *comparable* to other processors in these areas, its performance has suffered considerably due to the mismatch of address and data paths.

By considering the equivalent performance of a matched 32 bit processor, it can be seen that the ECS approach has in fact resulted in significant speed improvements over the SI and DI design paradigms. Furthermore, a low power implementation has resulted without explicitly designing for it, indicating that high speed asynchronous systems can

be targeted which still benefit from low power operation. Although unlikely to match the speed of a corresponding synchronous design, the difference can be greatly reduced by using ECS techniques in preference to SI and DI control schemas.

It is worthwhile noting that the area of *ECSTAC-32* is also considerably lower than the other asynchronous processors, all of which employ a pipelined control structure. Although in the case of AMULET this is partially due to the more complex ISA, it must be expected that a proportion of this deviation is also attributable to the reduction in complexity in the control circuits with a BD model. Therefore not only has the speed been improved (without sacrificing power dissipation), but so too has the area, and this in turn results in an important reduction in *cost*.

Chapter 8

ECSCCESS: A Superscalar Microprocessor

ALMOST all early microprocessors employed a pipelined approach in their implementation which separated the sequential requirements of instruction decode, operand fetch, execution, and the write back of results. This approach was convenient since the global register file (and scoreboard) helped control data hazards, and the instruction issue and execution order were maintained via the pipeline structure, which helped simplify the control schema.

However as the demand for higher performance continued, the restriction of in-order execution had to be abandoned. This is because the latency of an instruction to write back a result for subsequent use caused the execution of further instructions to be delayed, even if they weren't in conflict with the stalled instruction. Consequently the trend moved from pipelined to superscalar operation, in which the functional units (FUs) are operated in parallel, rather than sequentially. With such an architecture there is no longer a restriction on the instruction order (save that data hazards are still properly resolved) and the latency of any one unit is no longer compounded by the processing delay of other stages in the pipeline. This results in an overall increase in the system's throughput, and in fact almost every modern microprocessor now employs a superscalar architecture (however some degree of pipelining is often still used, either within each FU or in the stages prior to instruction execution).

The *ECSTAC* processor was constructed in a pipelined fashion primarily for ease of

implementation (as per the early synchronous microprocessor designs). However it became evident during the design that such a structure could in almost every instance be clocked faster than the asynchronous handshaking control could cycle between the stages, and as such it would be unlikely to ever exceed a synchronous implementation in terms of speed. Nonetheless, the advantage of reduced power dissipation is still highly favourable despite this speed deficit, as is the removal of clock skew management for ULSI processors.

A pipelined processor also cannot take full advantage of self-timed computations, since the system's throughput is limited by the slowest of these units. However, if asynchrony is to have any chance of outperforming synchronous systems in terms of speed, then the advantage of average case computation time must be fully utilized. This therefore requires parallelism of the self-timed units (rather than pipelining) which provides yet another reason for implementing a superscalar architecture.

Indeed, as the synchronous realm has switched from pipelining to superscalar operation, so too of recent has the asynchronous realm. While the early asynchronous microprocessors such as the Caltech design and AMULET employed a pipelined structure, those to emerge recently have utilized superscalar operation. Unfortunately however none of these have been fully implemented in VLSI, but have instead been implemented in VHDL with differing granularity (from gate level to functional blocks). This chapter describes the principle operation of these recent processors and then describes the design and implementation of the proposed *ECSCCESS* architecture (which is an acronym for **E**vent **C**ontrolled **S**ystems **C**PU **E**mploying **S**uper-**S**calarism). *ECSCCESS* was developed to fully exploit the potential speed advantages of both self-timed data computations and the ECS methodology.

8.1 Other asynchronous superscalar microprocessors

8.1.1 SCALP

The SCALP processor [End95b] was developed with the primary goal of reducing power dissipation rather than achieving high speed operation (although the latter helps this on a per instruction basis). Its general architecture begins with an instruction issuer, which is able to perform out-of-order and multiple instruction issues to the functional units (FUs).

There are four of these: ALU, memory, move, and register units, whose results are all sent to a router. A separate unit to handle instruction branches is also employed, but there is no facility for handling interrupts or exceptions.

There is no global register bank in SCALP. Instead, the router issues the results of each FU operation back to the source input of a subsequent operation (a destination FU is specified in the instruction word rather than a destination register). Since the majority of FU results are used only once, it was hoped that this approach would improve speed and reduce any unnecessary power dissipation associated with register storage. If a result is needed more than once, it can either be stored explicitly in the register bank, or replicated through the MOVE unit.

SCALP also employs its own ISA with a reduced code density, consisting of 12 or 24 bit instructions fetched in 64 bit chunks (with 4 control bits). Although this variable instruction length complicates the control schema as in *ECSTAC*, it is intended to reduce power dissipation per instruction.

The SCALP architecture's cycle time is limited by a long branch latency (241 gates [End95b]) and a reduced instruction rate for when a result is used more than once (often requiring additional instructions to copy it). However one significant cause of this was that explicit results forwarding (from the router) could only be used infrequently. Although most results are used only once, the FU which uses it is not always known at the time of the computation (being dependent on a subsequent branch). The SCALP architecture can only handle this situation by storing the result in the register bank, and then reading it out again for use when the required destination FU is known.

8.1.2 Fred

Fred [Ric96, RB96] is an asynchronous microprocessor which implements an ISA based on the Motorola 88100 processor. Only one instruction can be issued at a time, however if it stalls then a subsequent instruction can still be issued provided there is room in the instruction window to store it (and if it too doesn't stall). This approach still enables a regular rate of instruction issue even in the event of stalls (similar to SCALP in that an instruction stall doesn't necessarily halt the processor, although SCALP is also able to issue multiple instructions).

A global register bank is employed together with a scoreboard for handling data hazards. This enables the control schema to be kept simple but increases the execution latency over an explicit results forwarding mechanism (due to the store and load operations required in the register bank). Since most programs employ a significant dependency between adjacent instructions, this can slow down the processor speed and reduce the utilization of the parallel FUs.

There are four FUs employed in Fred which are used for arithmetic, logical, control, and memory operations. Branch detection and evaluation is decoupled from the FU instruction issuer, so that instructions can be issued in a variable delay slot until a subsequent *doit* command (after the branch) is encountered. Exceptions and interrupts are both supported in the VHDL model of the processor.

8.1.3 Rotary pipeline processor

The rotary pipeline processor (RPP) [MRW96] circulates results around a ring which is interspersed with a number of FUs and corresponding output latches. When the required source operands for a FU become available the FU triggers its operation, otherwise it passes the results through directly to the following stage of the ring (to the next FU). This effectively removes the global register bank and implements results forwarding as in the SCALP architecture, with a larger register bank similarly implemented as a separate FU.

The instruction issuer can provide simultaneous instructions to any or all of the FUs in the ring (again equivalent to SCALP), which can however be variable in number. Branch instructions can also be decoupled from the instruction issuer as in Fred which can therefore enable speculative operation. Interrupts and exceptions are not facilitated.

This architecture is similar to SCALP in that register forwarding is implicit, however it is non-restrictive in the number of FUs which can be connected into the ring. Although no performance measures are available, it appears that the essentially *pipelined* structure of the ring could cause excessive latencies, since a FU may have to wait for a result to propagate through the entire pipeline ring before being used. Furthermore pipelining a FU equates to placing extra stages into the ring which would further reduce the processor's speed.

8.2 Characteristics of *ECSCCESS*

It is evident that abandoning the global register bank is beneficial for reducing the latency between data hazards and also for reducing power dissipation, therefore in *ECSCCESS* the register bank is implemented as just another FU. Although SCALP implements this via results forwarding through the router, its functionality is limited in that only one FU can receive its value, so that in the majority of cases the result must still be routed into the register bank.

ECSCCESS overcomes this problem by removing the router. Instead, each result from a FU is placed onto its own global output bus and *remains* there until overwritten by a subsequent instruction. As such it is possible for any number of latter instructions to any FU to source its value immediately after being produced. The result only ever needs to be written to a register if more references to its value are required after an instruction which overwrites it (although the MOVE instruction described later provides an alternative to this).

One other problem of SCALP (and Fred) is that the number of FUs is fixed, which prevents additional parallelism from being incorporated. Although the RPP overcomes this problem, it does so at the expense of an increased latency for stalls. In *ECSCCESS* the number of FUs for a given implementation can be anywhere from 0 to 32. Furthermore, their functionality is non-specific: 32 ALUs could be used; or just 1 ALU and 1 memory unit; or whatever. There doesn't even have to be a register bank FU (although this would undoubtedly be useful). Unlike the RPP, the latency for data hazards is not influenced by the number of FUs implemented, as will be seen by the architectural description in Section 8.4.

In all of the processors discussed above an instruction which stalls does not necessarily prevent other instructions from being issued. In SCALP and the RPP the instruction issuer only stalls if for the fetched block of instructions (given that multiple instruction issue is possible) every target FU is stalled, and in Fred this occurs when the instruction window fills, implying that all instructions therein are waiting for results to compute. *ECSCCESS* implements a technique similar to Fred, however instead of an instruction window each FU is preceded by a small FIFO of length n . The instruction issuer will therefore only stall if n instructions have already been issued to the same FU, all of

which are stalled (or if $n + 1$ instructions have been issued with the first having an extremely long computational latency within the FU). Assuming 32 FUs with $n = 3$, this implies a maximum of 97 instruction issues before stalling and a minimum of 4. Multiple instruction issue is not implemented in *ECSCCESS*, nor are exceptions (although Section 8.11.2 describes how these can be incorporated into the architecture).

8.3 Instruction set architecture

The full ISA of *ECSCCESS* is detailed in Appendix C and only a brief summary is given here. A 32 bit data and address path is employed, and every instruction is encoded in one word.

The JUMP and JCOND instructions (collectively termed JMP instructions) enable unconditional branches and branches dependent on the result of a previous operation (typically a comparison). A CALL instruction enables a subroutine to be entered which is concluded with a RETC instruction (termed SUB instructions, and together with JMP are termed branch instructions). Each of these instructions branches relative to the PC, either using a signed 28 bit offset encoded in the instruction word or a value located on a specific FU bus.

ECSCCESS implements an on-chip stack which stores the return address for RETC instructions and interrupt and trace routines, as well as the process status register (PSR) for the latter. This stack can be any length, however in the event that it eventually fills the stack must be continued off chip in main memory. The LDSP instruction enables the start address of the off-chip stack to be specified in the SP register.

A 7 level prioritized interrupt facility is implemented, with the highest level (7) being non-interruptible. When an interrupt occurs with a priority level (IPL) higher than that currently stored in the processor (in the PSR), then an interrupt routine starting at the address pointed to by $IVR + IPL$ in memory is executed. The interrupt vector register (IVR) can be set with an IVRL instruction to point to a set 32 byte block containing the interrupt table. Instructions are also available to enable and disable the interrupt facility, return from an interrupt, and to set the current IPL of the processor.

A trace mode is also available. When enabled (through an ET instruction) a trace routine starting at the address pointed to by the IVR is executed after *every* instruction. This can be helpful for customizing the analysis and debugging of instruction streams.

There are four basic FU instructions: MOVE, LDC, FUbus, and FUimm. The MOVE instruction transfers the value from one FU bus onto another, bypassing the FUs activation. This enables values computed from one FU to be transferred to another for iterative use (such as loading the variable a from memory, *moving* it to an arithmetic unit, and executing a loop of $a = a + i$). The LDC instruction enables an encoded constant to be moved onto a FU bus. Furthermore, various status registers such as the PC, SP, PSR, and IVR can also be loaded onto a FU bus for manipulation.

The FUbus instruction activates a FU with zero, one, or two source FUs whose bus values provide the arguments for the function, and a FUimm instruction encodes a constant value in place of the second argument. The actual set of instructions which are available within each FU is implementation dependent, since the number and type of FUs in the architecture is variable. However, the proposed FU allocation given in Appendix C for *ECSCCESS* provides the following FUs and corresponding instructions.

A register bank FU is used with 4 output buses onto which the register contents can be read. It is unique for a MOVE instruction in that a source value can be written into a register as well as being transferred to the specified output bus. The large 16 bit register field enables a hierarchy to be conveniently implemented within the register bank.

Various integer arithmetic units enable addition, multiplication, shifting etc., as well as a variety of floating point operations (although specific instruction sets for these have not been specified).

A specialized comparator unit is also implemented which compares the values of two buses and sets the MSB (branch bit) of its output bus accordingly (which is then used to govern conditional branches). The comparator can also implement addition and subtraction operations, with the result being placed onto the output bus and a collection of flags stored in the internal FR, which gets read out with the branch bit for a compare instruction. Another compare instruction is implemented which sets the MSB according to the current state of the FR.

The memory unit performs load and store operations, and also has the ability to modify the memory address of the last memory operation for use as the address for the current one. This enables array indexing to be performed without having to use the other FUs. Finally, there is also support for an integer and floating point co-processor to be used.

8.4 General architecture

The general structure of *ECSCCESS* is shown in Fig.8.1. The instruction issuer (termed the *sun*) constantly cycles with the IC (if present) fetching a new 32 bit instruction on each cycle. Since the ISA employed in *ECSCCESS* is based on FUs rather than instruction types (although the two are often similar), it is possible to decode all of the relevant control signals from the instruction opcode in just one gate delay. If a FU instruction is occurring then it is sent immediately to the *globe*, together with the value of the *constant* bus (required only for a LDC instruction) and the *opcode* required by the FU to specify its operation. This process will not stall unless the preceding FU instruction was not able to be loaded into its FIFO due to it being already filled with stalled instructions.

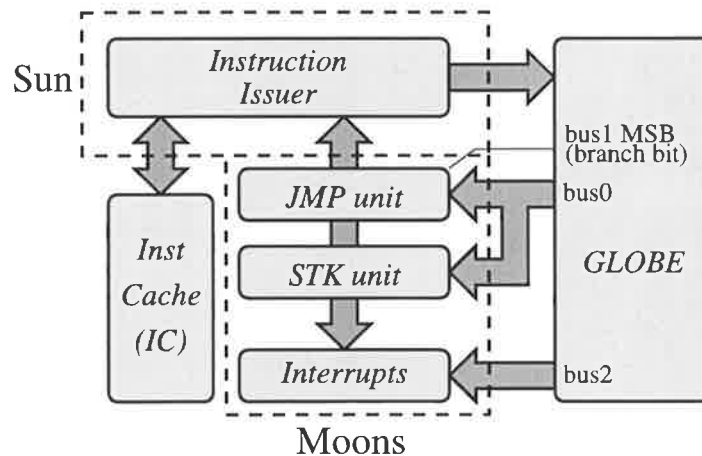


Figure 8.1: General structure of the *ECSCCESS* microprocessor.

The globe contains each of the FUs and their respective output buses, and is constructed as shown in Fig.8.2. Each FU has a unique output bus to which it writes its results (although some FUs such as the floating point and the register units have more than one output bus), and is able to read a result from any other FU's output bus (these buses are collectively termed the *ocean*). RAW and WAR hazards are handled within each FUs preceding control block (termed the *shore*), which also multiplexes in the appropriate source values for the FU as well as implementing the FIFO. WAW hazards are handled implicitly by the in-order execution of each FU. The control structure of the shore is rather complex, and is discussed in detail in Section 8.5. Note that regardless of how the FU operates, the control schemas for each shore are identical.

If the globe is not triggered by the sun, then either a branch or interrupt instruction

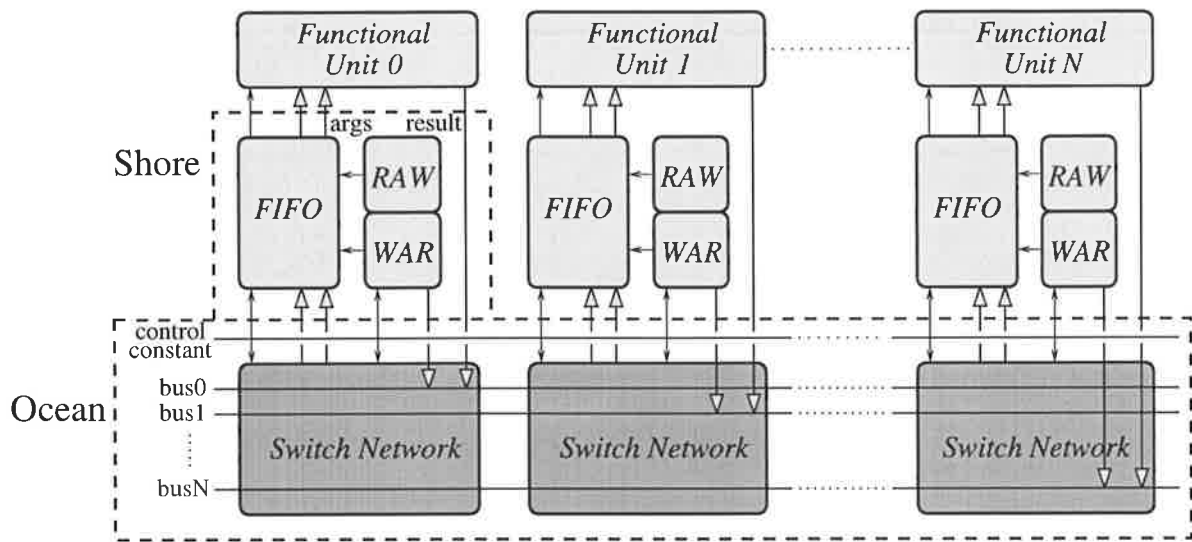


Figure 8.2: General structure of the globe.

(not an interrupt itself) has occurred. These units are collectively termed *moons*, and their operation is closely related to the sun and is described in detail in Section 8.6. Note that the globe does in fact have a minimal interaction with these units, since *bus0* is used for branch-bus instructions, *bus2* provides the new PC location from memory for an interrupt routine, and the MSB (bit 31) of *bus1* provides the result of a comparison which is used for a JCOND instruction. Typically FU2 will be the memory unit (*dest* = 15 from the allocation in Appendix C), FU1 will be a comparator unit (*dest* = 12), and FU0 an arithmetic unit (*dest* = 7).

8.5 Implementation of the *shore*

The shore is required to implement all of the control structures for resolving data hazards as well as for selecting the relevant FU and multiplexing in the appropriate source values. The shore also implements a FIFO buffer prior to the actual activation of each FU to reduce the possibility of processor stalls.

The resolution of WAW, RAW, and WAR hazards is a complex problem since it's possible for many FUs to be active at once. WAW hazards are handled implicitly by the in-order operation of each FU and its preceding FIFO, since if one instruction stalls at the input to the FU then a following instruction in the FIFO will be forced to stall as well until the former has begun execution. Controlling RAW and WAR hazards however

is rather more complex, especially since the inclusion of the FIFO increases the amount of potential parallelism.

8.5.1 Controlling RAW hazards

The control pertinent to RAW hazards is shown in Fig.8.3. The sun supplies the input event ∂fu to every FU in the globe. The relevant one to be activated is decoded from the FU field in the instruction word (fu_isthis), which is then used to mask the activation of the FU ($\partial fu0$). Once initiated into the first stage of the FIFO, the event $\partial fudone0$ is returned which is used to increment the input counter $Crawin$. Note that using $\partial fu0$ for this would require an additional state in the counter (and therefore a three bit data width). The input (and output) counter values for every FU are placed onto the ocean. At a later time the instruction will complete in the FU, and $selout$ will pulse high to latch the result of the computation onto the output bus (which also goes to the ocean), and the output counter will then be incremented.

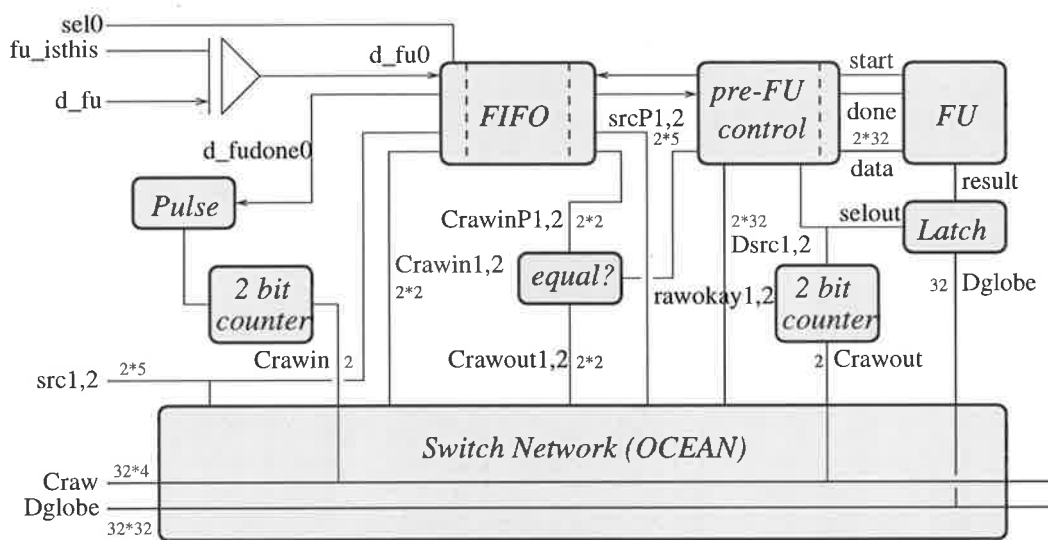


Figure 8.3: Control structure for governing RAW hazards.

Conversely, an instruction which sources a FU will first grab the relevant $Crawin$ value for that FU from the ocean, and proceed through the FIFO (if possible). Note that if the instruction sources and executes from the same FU, then a counter value of 0 is read (since the hazard will then be resolved in the same way as a WAW). Furthermore, the minimum cycle time of the sun is such that the $Crawin$ value from a previous instruction will be valid before a subsequent instruction attempts to source it.

At the output of the FIFO, and prior to the activation of the FU, the output counter value for the source FU C_{wout} is selected and compared against C_{winP} , which is the original value of C_{win} after propagating through the FIFO. If $C_{wout} \neq C_{winP}$ then the required FU source values haven't yet been computed, and the instruction will stall, otherwise if they are equal then the relevant bus data will be switched onto the source bus D_{src} and the FU activated.

Each counter needs to be wide enough to ensure that C_{wout} cannot overtake C_{winP} (and therefore incorrectly indicate valid data to a subsequent sourcing instruction). Since there can be three instructions present within each unit at any time (as seen by the dashed lines in Fig.8.3 which indicate latching elements), a 2 bit counter (with 4 states) is sufficient to prevent this.

8.5.2 Controlling WAR hazards

WAR hazards are controlled in a similar method to RAW hazards, except in this case the input counter is triggered for a FU if it's required to be sourced, and the corresponding output counter is triggered once the sourcing instruction has grabbed the data and begun execution. The control schema used is shown in Fig.8.4.

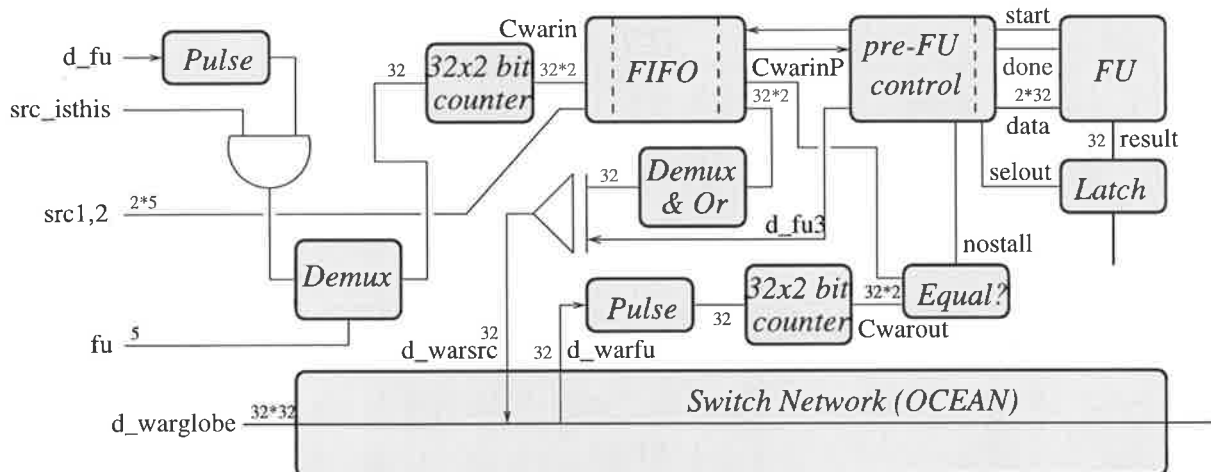


Figure 8.4: Control structure for governing WAR hazards.

The WAR control can be essentially split into two phases. If an instruction to FU_j needs to source FU_i , then C_{win_j} of FU_i will be incremented. When a subsequent instruction to FU_i takes place, the counter values will proceed through the FIFO (to C_{winP_j}) and be compared against the counter values of C_{wout_j} .

Once the first instruction to FU_j has sourced the data from FU_i (as given by the signal $\partial fu3$ from the pre-FU control block), an event is sent back through the ocean to FU_i . This then generates a pulse which increments the $Cwarout_j$ counter within that unit.

Before an instruction to FU_i is initiated, the $Cwarout$ and $CwarinP$ values for each FU are compared. If these are all equal, then there are no preceding instructions which are still waiting to source the data, and the instruction can proceed ($nostall = 1$). Otherwise, if $Cwarout_j \neq CwarinP_j$ for any j , then FU_j still needs to source the old data from FU_i and so the instruction is stalled. Note however that in practice the computation within the FU is still initiated regardless of the state of $nostall$, however only when this is high is the result written onto the output bus (ie- the $selout$ pulse is enabled).

8.5.3 Structure of the pre-FU unit

The pre-FU control unit is used to halt the activation of the FU until all source data is available (as given by $rawokay1, 2$ in Fig.8.3), generate an activation pulse for the self-timed operation ($start$), and stall the writing of the result until all sources of the previous bus value have occurred (as given by $nostall$ in Fig.8.4). The structure of this unit is shown in Fig.8.5.

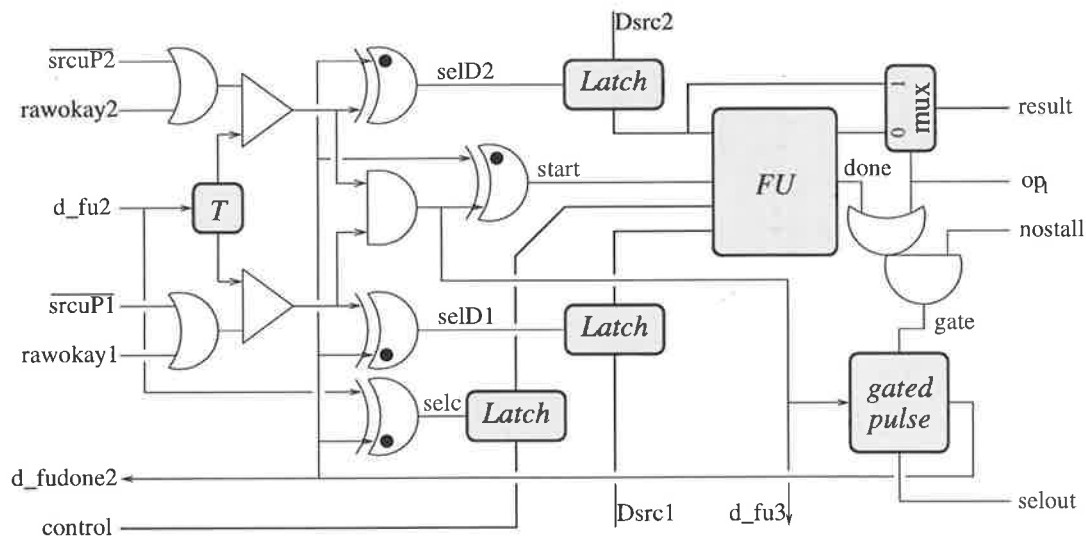


Figure 8.5: Control structure for governing the operation of the FU.

Firstly, the input event from the FIFO ($\partial fu2$) is delayed until the relevant counter signals have been switched in from the ocean and compared to give the $rawokay$ signals. If this is high for $src1$, or if no bus data is needed for this operand (as given by $src1u = 0$),

then an event is passed through the *send* gate which causes the source data to be latched. Similar arguments apply to *src2*. When both operands contain valid data, the self-timed FU is activated via $\partial fu3$ and the *until* gate. This event is also sent to the WAR control to indicate that the required source data has been latched.

At the conclusion of the ST operation *done* will go high, however if a MOVE or LDC instruction is occurring (when $op_1 = 1$) then *Dsrc2* is multiplexed directly to the result and the *done* signal from the FU is ignored. Provided that $nostall = 1$, the result of the operation will be latched onto the output bus, otherwise it will be stalled until this occurs. A gated pulse circuit is used to generate the *selout* signal which latches the data, from which the done event $\partial fudone2$ is also produced. This event resets the ST FU by setting $start = 0$ and also opens the latches for new data to arrive (it is assumed that the reset time to $\nabla done$ is less than the cycle time to the next $\partial fu3$ event). This event is then also sent back to the preceding FIFO to fetch the next instruction in the queue.

8.5.4 Generating the return event to the sun

From Fig.8.3 it can be seen that the latch select signal *sel0* of the first stage of the FIFO is returned to the sun, rather than the acknowledge event $\partial fudone0$ from the state pipeline control. If the latter were returned, then a single acknowledge event would have to be generated by *or*'ing the 32 $\partial fudone0$ events from the FUs (assuming 32 such units are used). This involves a 5 level tree of *xor* gates and would severely reduce the potential cycle time of the processor. Instead, the control circuitry of Fig.8.6 is used.

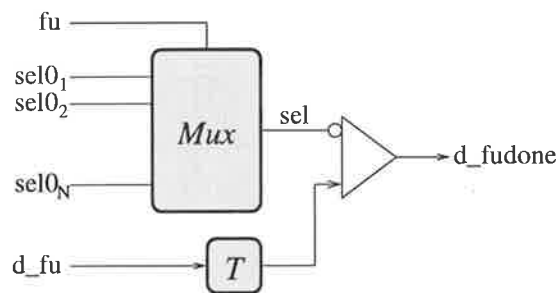


Figure 8.6: Generating a return event to the sun from 32 FUs.

The relevant latch signal is first selected through the multiplexer, and if this is still high then the FIFO has been stalled and the event ∂fu is kept pending at the *send* gate. This event is delayed slightly to occur after the selection (which occurs in parallel with the

FIFO triggering and has no effect on the cycle time). If however $sel = 0$, then the data has been latched into the FIFO and the return event is generated in just $t_{mux} + t_{inv} + t_{send}$, as opposed to $5t_{xor}$ for the merged events scenario. It should also be noted that the slower, merged events approach is SI, whereas that presented in Fig.8.6 is BD.

8.5.5 Switching network

The switching network is used to multiplex the data values from the ocean onto the source buses for the FU, as well as selecting the corresponding *Craw* signals for managing RAW hazards. In practice drivers are used instead of multiplexers for reading the source data, as this enables it to become valid earlier than the 5 level tree of multiplexers which would be needed for 32 FUs.

This unit is governed by control signals from the shore (in particular, *src1,2* and *srcP1,2* before and after the FIFO), and given that $2 * 32 * 32 = 2048$ drivers could be needed per FU (just to multiplex the source data), it has the potential to be quite large and perhaps inhibitive to the overall architecture. Section 8.8 analyses this possibility and shows how it can be floorplanned in a reasonable size.

8.6 Implementation of the *sun* and *moons*

The sun is required to trigger the globe or the moon(s) with the appropriate control signals, as well as updating the PC in preparation for the next instruction fetch. However, to enable the rapid execution of both the globe and the moons, these are all conditionally activated (through *feed* gates) from the return event ∂_{cpu} of the IC, rather than through a tree of *select* gates from the sun. This enables a much faster processor cycle time and a lower latency for branch evaluation. The overall structure of the sun and moons is given in Fig.8.7, with the general functionality of each unit described thereafter (except for the decoder unit, which simply generates control signals to the other units from the *op* field of the instruction word, and operates in just one gate delay). Note that the interrupt controller is not shown in the diagram, but is discussed in Section 8.11.1.

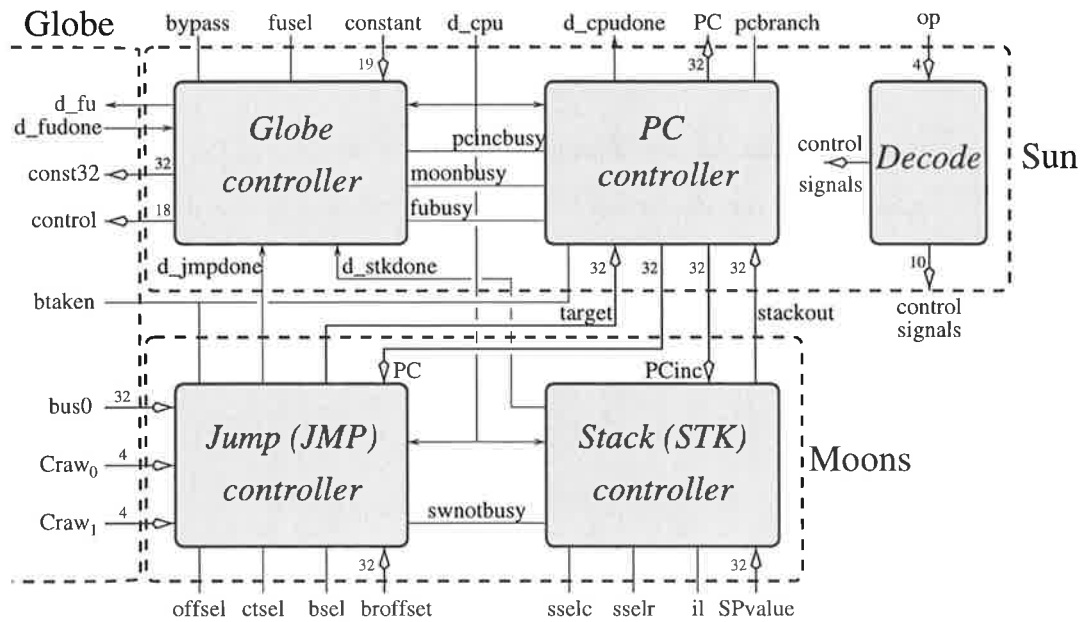


Figure 8.7: Overall control structure of the combined sun and moons system.

8.6.1 Globe controller

This unit forms part of the sun and is used to interface to the globe, as well as providing control signals to the PC controller for when the globe or a moon is still busy, which are then used to stall the refetching of a new instruction. Its configuration is shown in Fig.8.8.

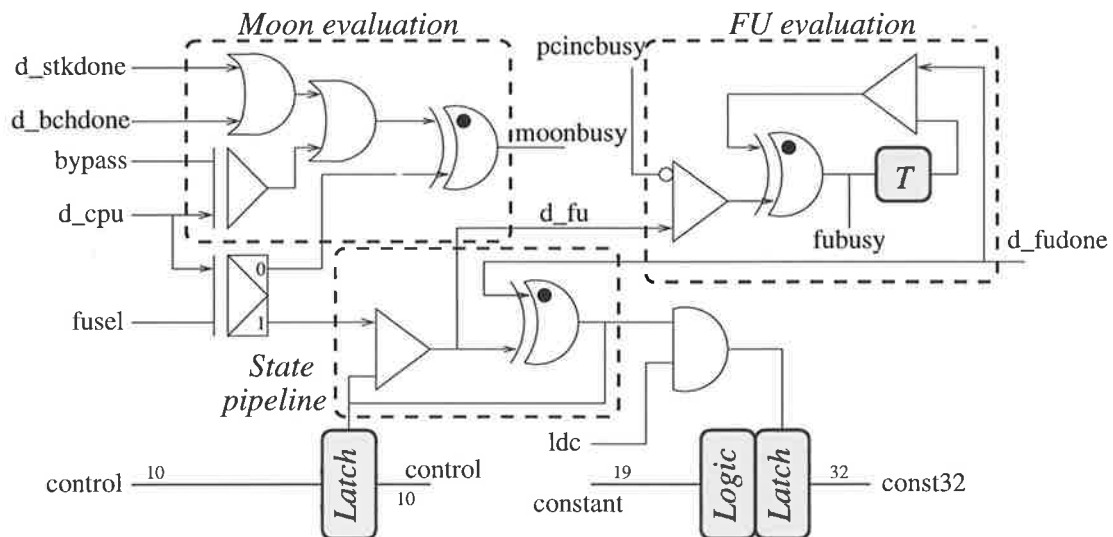


Figure 8.8: General structure of the globe controller.

The event ∂_{cpu} from the IC is steered into the Moon evaluation block if the globe is not activated by this instruction (when $fusel = 0$), and sets the signal $moonbusy$

high until the relevant moon (JMP, STK, or neither if *bypass* = 1) has completed its operation. Otherwise, if the globe is activated then a state pipeline controller is initiated which latches the control data onto the globe as well as the new value of the *const32* bus (for a LDC instruction only). The ∂fu and $\partial fudone$ events to and from the globe are used to determine the state of *fubusy* (essentially through an *until* gate), however the signal *pcincbusy* is also used to prevent $\Delta fubusy$ from occurring until the PC has incremented from the previous instruction. Although this is not hazardous, if the globe also stalls then without this circuitry the PC controller will unnecessarily wait for the globe to *uninstall*, which is otherwise postponed to the next instruction and enables an earlier IC fetch (maximizing parallelism).

8.6.2 PC controller

The PC controller of Fig.8.9 essentially consists of two components. Firstly, the signal *pclatch* is generated from a gated pulse circuit, after which the event $\partial cpudone$ is sent back to the IC to fetch the next instruction. The gate to this circuit goes high when the new PC value is valid, which is either when the increment of the PC for the previous instruction has completed for a globe instruction, or else when the target has been computed or fetched for a program branch which is taken.

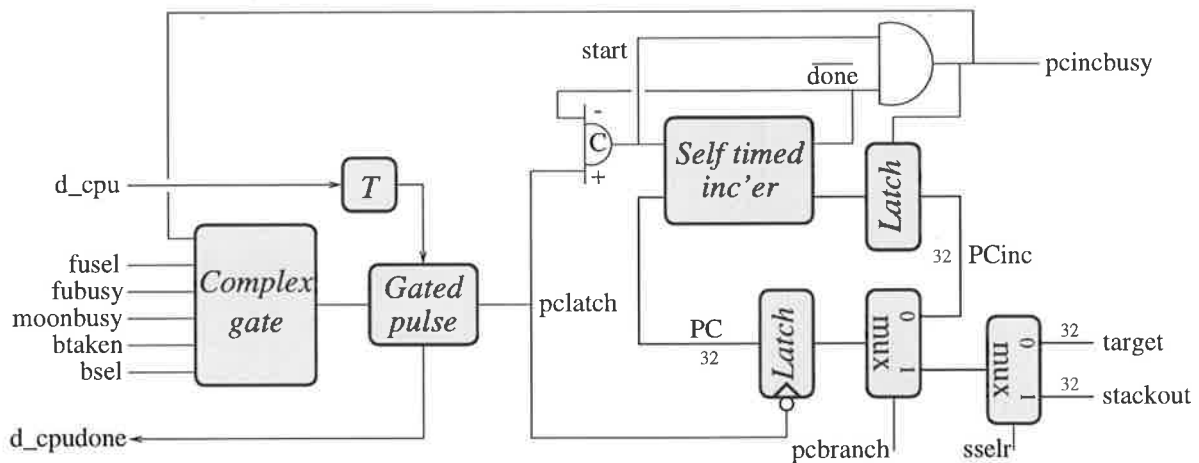


Figure 8.9: General structure of the PC controller.

The second component of the controller initiates the incrementing of the PC value as well as latching the new PC value for the next instruction when $\Delta pclatch$ occurs. The incrementer is initiated when *pclatch* = 1 and its result is latched when *done* = 1 (which

also resets the ST incremter to complete the cycle). Since the first instruction at PC=0 has no predecessor to initiate the increment of the PC ready for the next instruction, the PCinc bus is initialized to “1”.

8.6.3 Branch moon controller

A PST adder is used to compute the branch target, and is initiated from ∂_{cpu} (if needed) immediately unless the value of *bus0* is required for the PC offset, in which case the addition will be stalled until the relevant FU has computed its result (when $Crawin_0 = Crawout_0$). Fig.8.10 shows the control circuitry used to effect this.

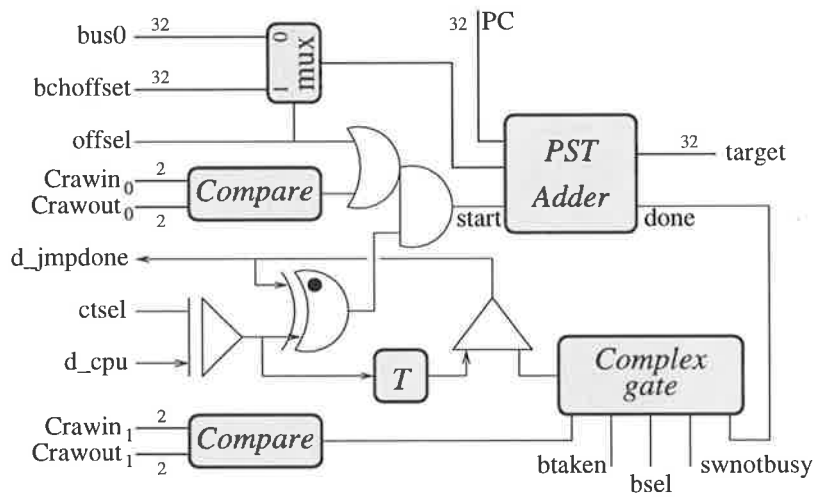


Figure 8.10: General structure of the branch unit.

Furthermore, a conditional branch cannot indicate completion (giving $\partial_{jmpdone}$) until the branch flag of *bus1* has been set, which is indicated when $Crawin_1 = Crawout_1$, although the adder can still be initiated and its result ignored if the branch is not taken. Note also that the return event $\partial_{jmpdone}$ is stalled if *swnotbusy* = 0, which indicates when the address for a RETN instruction (PC+1) has been stored onto the stack. Section 4.3.1 explains this technique, which can be used here since a stack load always occurs concurrently with a branch target computation.

8.6.4 Stack moon controller

Fig.8.11 shows the control circuitry for executing the stack operations for LDSP, RETN, and CALL instructions. The first two instructions are triggered in the stack when ap-

appropriate (via ∂ls and ∂rs respectively), and once the SP has been loaded (or the RETN address *stackout* fetched) the output events are generated and merged to give $\partial stkdone$. A CALL instruction is stalled at the input to the stack until the return address *PCinc* has computed, and *swnotbusy* is sent to the branch moon to govern the generation of its return event $\partial jmpdone$.

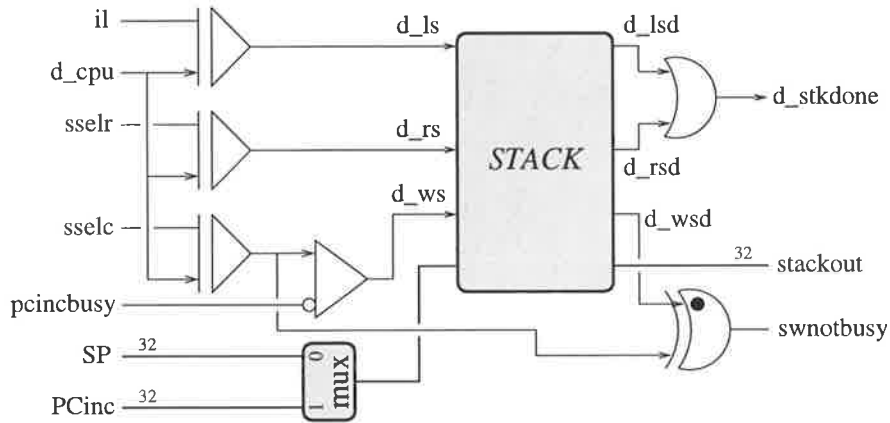


Figure 8.11: General structure of the stack unit.

The stack itself is implemented with a separate register for the SP, and a register structure with counters for the write and read phases governed by ∂ws and ∂rs respectively. The length of this stack is implementation dependent, and once filled must execute a subsequent write operation to the memory address pointed to by the SP. Clearly, the larger the stack size the less frequently will such external memory operations be required.

8.7 Implementation of functional units

The FU allocation table in Appendix C provides for numerous different FUs to be incorporated into the architecture, and proposed implementations for some of the integer units are described in the following sections.

8.7.1 AID unit

This unit can be implemented simply with a ST incremter and a PST adder operating in parallel. To enable negation and decrementing, the incremter should have its input and output *xor*'ed with the relevant control signals (since $x_{dec} = \overline{\overline{x}_{inc}}$ and $-x = \overline{x}_{inc}$).

Flags get computed and stored in an internal FR during the reset (precharging) phase between activations, and can be read out with a subsequent FLAG instruction.

To minimize the control overhead both ST units should be activated by *start* unconditionally, and the *done* signal for the appropriate unit masked at the output. This essentially combines the *and* and *or* gates which would otherwise be used (before and after the ST units) into a single, faster complex gate at the output, however this marginal speed advantage may be outweighed by the additional power consumption for other FUs.

8.7.2 MEM unit

The implementation of the memory unit (ignoring the complexities of the DC) merely consists of an incrementer and a decremter in parallel. If a memory operation with $UL = 0$ is occurring, then the address from *arg2* is placed onto the address bus as well as being latched into the LMA (last memory address) register. As soon as this (or any other) operation is complete, the inc and dec units are activated to provide LMA+1 and LMA-1 for the next operation. If $BA = 0$ then the inc/dec result (as given by the ID field) is multiplexed onto the address bus prior to the DC being activated, otherwise the original LMA is used, and at the conclusion of the operation the inc/dec result is latched into the LMA as above.

8.7.3 CMP unit

This unit consists of a ST comparator and a PST adder. Both units are activated from *start*, and for the comparator the MSBs of the two operands must first be compared and masked if a signed operation is occurring (since the comparator of Section 6.5 is for unsigned numbers). The branch bit of the output bus is then set according to the result of this comparison together with the current status of the FR.

An ASC instruction performs an addition or subtraction and places the result onto the output bus. Flags are computed from the adder's result and stored in an internal FR. A comparison is also initiated, the results of which are also stored in the FR. A subsequent CMP0 instruction will then set the branch bit according to the state of any one of these flags.

8.8 Floorplanning issues

Before proceeding with the actual implementation of *ECSCCESS* it was necessary to evaluate the practicality of such an architecture in terms of its area usage. Clearly, if no floorplan for the processor could be devised with a reasonable aspect ratio (say, $\approx 15\text{mm}$ each side) then the architecture must be revised, or abandoned. Such an investigation is vital given that *ECSCCESS* has only been specified and simulated in VHDL. As a guideline for evaluation the ES2 technology was assumed.

8.8.1 Size of the ocean

Firstly, the size of the ocean must be considered, since this contains a large array of both data and control signals and could therefore be the governing factor for the VLSI floorplan. In the architecture as presented, each FU places 32 bits of results data, 32 events for WAR control, and 4 bits of RAW control data onto the ocean, totalling 68 wires per FU. Furthermore, the *constant* bus of width 32 (used for LDC instructions) is also placed onto the ocean from the sun.

In the ES2 technology, the width and spacing required for a wire (including space for contacts from the FU) is $28\mu\text{m}$, resulting in an ocean width of $w_{ocean} = (32 * 68 + 32) * 28\mu\text{m} = 6.2\text{mm}$. Therefore the ocean will undoubtedly be a governing factor in the floorplanning of *ECSCCESS*.

8.8.2 Size of the switching network

Another contributing factor to the area and floorplanning of the processor is the area required for the switching network (which must be replicated for each FU). Fig.8.12 shows the structure of a dual driver component which can be replicated across and along the ocean to comprise the switching network. Note however that a 3-layer metal process would be needed for such an implementation (one layer for the ocean, one to orthogonally route back the data values, and one to supply power, control, and metal routing to and within the circuitry).

The router feeds *src1*, *src2*, *srcP1*, and *srcP2* (and their inverses) across the globe to the integrated decoder (5-input nand gate), inverter (to generate its complement), and

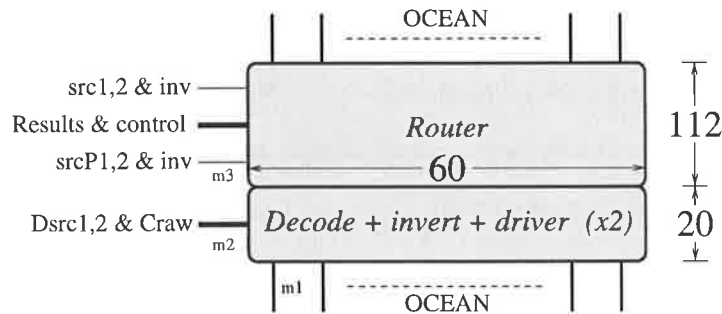


Figure 8.12: A driver component used in the switching network with dimensions quoted in μm .

driver circuit. Two such driver modules can be designed in the ES2 technology in the area as quoted on the figure.

With a $60\mu m$ width a total of $6200/60 = 103$ driver pairs can be implemented across the width of the ocean. Given that 32 individual drivers are needed per bit, a total of $103 * 2/32 = 6$ (integer part only) data signals can be returned to the FU in one layer of drivers. In a separate layer of metal, this would require $6 * 2.8 = 17\mu m$ which is less than the $20\mu m$ height of each decoder layer. Note that the same router can be used for multiple adjacent driver units, placed in equal quantities on either side of the router to minimize the polysilicon path lengths.

A total of 68 signals (64 data and 4 RAW control bits) need to be sent to the FU, which means that $68/6 = 12$ layers of drivers are needed to provide all of the source data. This results in the height of the switching network along the ocean of $h_{sn} = 12 * 20 + 112 = 352\mu m$. Furthermore, 68 wires also need to be routed back onto the ocean, requiring an additional height of $68 * 2.8 = 190\mu m$. However, assuming that vias between the second and third layers of metal are possible then this can be overlapped with the height of the router, resulting in a total minimum height for the switch network and route back block of $h_{snrb} = 352 + (190 - 112) = 0.43mm$.

Given that the width of the ocean and the minimum width of a FU (equal to h_{snrb}) are now known, it is possible to investigate potential floorplans for *ECSCCESS*.

8.8.3 A floorplan based on the minimum FU width

A general floorplan for *ECSCCESS* based on the minimum width of each FU (as given by h_{snrb}) is shown in Fig.8.13a. The FUs are placed evenly on either side of the ocean,

resulting in a total possible width per FU of $2 * 0.43 = 0.86mm$.

Assuming that the height of each FU is a generous $h1 \approx h2 = 1.4mm$, and that the height of the sun and moon circuitry is $\approx 0.5mm$ (which is reasonable given the large width of this block), the width of the processor core can be estimated at $w_{core} = 9mm$. The height of the core will depend on the number of FUs which are used. For the maximum of 32 FUs the height will be $h_{core32} = 14.3mm$, and for the 18 FUs used in the proposed allocation in Appendix C the height of the globe is $h_{core18} = 8.3mm$. Therefore the size of the processor as presented is certainly small enough for single chip fabrication.

However, the area will increase if an IC and DC are also used (as would be expected). Assuming widths of 3.5 and 1.5mm respectively (and lengths as long or wide as the processor depending on their orientation), the floorplan for 32 FUs would have the IC to the right of the processor and the DC to the left, resulting in a total area of $14 * 14.3mm$ (width*height) which is still small enough for fabrication. For 18 FUs, the IC would instead be placed below the processor, resulting in a total area of $10.5 * 11.8mm$. Note that the aspect ratio of both of these floorplans is close to one.

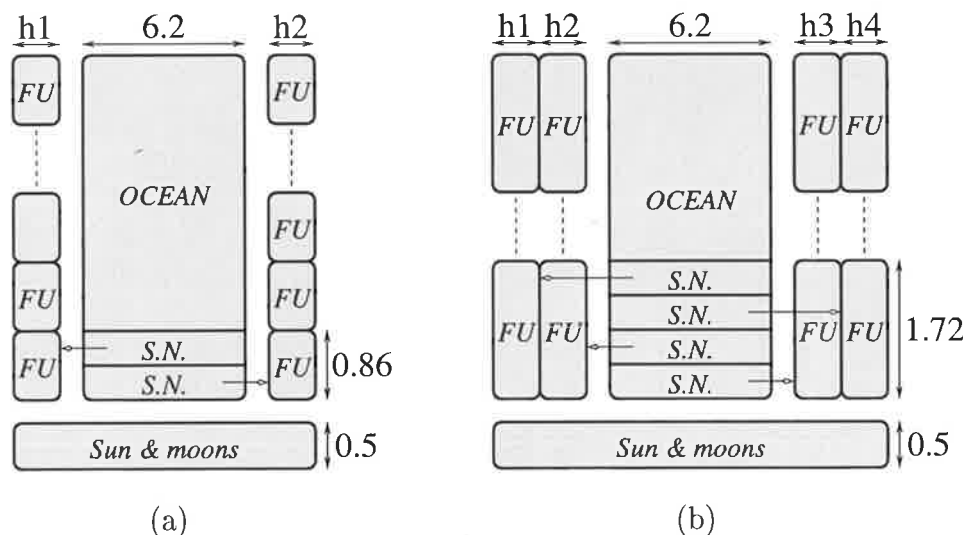


Figure 8.13: Floorplans for *ECSCCESS* based on (a) the minimum width of a FU, and (b) a width twice as large. All dimensions are quoted in *mm*.

8.8.4 Floorplanning for a larger FU width

A minimum FU width of $0.86mm$ provides for a single data path width in the FU of at most $860/32 = 27\mu m$. Although this width may be reasonable for some FUs (enabling approximately 3-4 adjacent gates in the ES2 technology), other applications may require

a wider data path if this cannot be compromised against the height of the FU.

Assuming then a data path twice as wide, the width of a FU (equal to 4 driver widths) would be $1.72mm$. Implementing 32 FUs across the floorplan of Fig.8.13a would result in an excessive height (up to $28mm$), therefore the floorplan of Fig.8.13b could be used instead, in which two layers of FUs are used on each side of the ocean. Note however that this would require the source and results buses of the second FU to route across the first (either in a separate metal layer or feeding alongside each single data path of the first FU).

For such a structure (assuming heights of each FU as before) the width of the processor core would be $w_{core} = 11.8mm$ with heights of $h_{core32} = 14.3mm$ and $h_{core18} = 8.6mm$. The 32 FU implementation would place the IC to the right of the processor core and the DC above it, resulting in a total area of $15.3 * 15.8mm$, which is only slightly larger than the area used assuming minimum FU widths. The 18 FU implementation would instead place the IC below the processor core and the DC to the right, giving an area of $13.3 * 12.1mm$, also only marginally larger than before.

It is argued then that the *ECSCCESS* architecture is feasible for VLSI fabrication in the ES2 technology. Clearly, the area usage of *ECSCCESS* will be even less in the smaller process technologies which are currently in use.

8.9 Simulation results

An implementation of *ECSCCESS* was constructed in VHDL at the *gate* level, and incorporated four integer FUs in total: a memory unit, a comparator unit, and two arithmetic units. Although a more practical implementation would also have a register bank, multiplier, and floating point units, this basic implementation of *ECSCCESS* still enables a significant amount of parallelism and even more data dependencies.

Random instruction streams were used for testing *ECSCCESS* since developing a suitable compiler would have been too time consuming. Given that only 4 FUs are implemented, a random instruction stream will still exhibit numerous data hazards comparable to those which would be present in most compiled programs anyhow.

Four different instruction streams were produced for testing various aspects of the architecture: one which incorporated all possible instructions; one which executed only

those instructions utilizing the FUs; one which executed 70% branch instructions (and the rest LDC instructions); and one which executed 35% call instructions. The instruction-type frequencies which were used for the all-instruction stream generation are shown in Table 8.1 (based on the information contained in [HP90, Chapter 4], from which the frequencies for the other streams can be deduced. The FUproc and FUspec instructions were distributed as approximately 15% comparator, 25% memory, and 30% per arithmetic unit (which correlates to the ratios used for *ECSTAC*), and 1 in 7 instructions (on average) executes a potential program branch.

Jump		Branch		Call (Retn)		Stack		FUproc		FUspec	
Joff	2.1%	Boff	8.6%	Coff	2.1%	LDSP	0.2%	FUimm	25.7%	LDC	4.3%
Jbus	0.2%	Bbus	0.9%	Cbus	0.2%			FUbus	51.4%	MOVE	4.3%

Table 8.1: Instruction frequencies used in generating code for *ECSCCESS*.

Furthermore, for the all and FU only instruction streams four different types of hazard minimization were investigated: no minimization; minimize RAW hazards only; minimize WAR hazards only; and minimize both RAW and WAR hazards. These last three types also minimize WAW hazards if possible since these can also reduce the processor's speed by increasing the stall time of the RAW and WAR hazards. The other two types of instruction streams implemented only the full minimization of hazards.

The simulation results for the *ECSCCESS* processor are given in Table 8.2, for which the Mips (5) and Mips (10) fields imply a DC with a 5ns and 10ns cycle time respectively (both this and the IC are assumed to have a 100% hit rate). All values quoted are averaged over 10 instruction streams of at least 1000 instructions each.

Instruction stream	Hazard minimization	Mips (5)	Mips (10)
All instructions	no minimization	117.4	113.8
	WAR and WAW	127.8	124.9
	RAW and WAW	140.6	140.6
	All minimized	141.7	138.8
FU only	no minimization	118.9	112.8
	WAR and WAW	138.7	133.0
	RAW and WAW	181.0	177.3
	All minimized	180.5	172.7
Branch and LDC	All minimized	84.8	84.9
Call and LDC	All minimized	138.7	139.8

Table 8.2: Simulation speeds of *ECSCCESS* for varying DC times.

Each complex gate delay was assumed to be 1ns except where a combination of gates with a low load was implemented in which case the overall delay was assumed to be 1ns (such as for a *nand* and *mux* combination with the *mux* output driving only one gate). It is expected that these delay assumptions will give overall delays comparable to those of the ES2 technology when implemented at the mask level.

It is evident from this table that the low Mips for branch instructions is the governing factor in the Mips difference experienced between the all and FU only instruction streams. This is expected since a branch must wait for a preceding comparison to execute as well as the branch target to be calculated (which will only marginally affect this). Therefore to reduce the speed deficit of branch instructions a compiler ought to place a useful, non-dependent instruction between the comparison and the branch if possible (an optional delay slot). Note however that when no hazard minimization is implemented the effect of branch instructions is negligible anyway. This is because the dependencies between FUs is governing the cycle time of the processor, and the branch detection mechanism operates in parallel with this.

Furthermore it can be seen that implementing hazard minimization can have a considerable effect on the performance, improving the Mips by up to 20% for the all-instruction type. Therefore a compiler ought to place a significant emphasis upon this, and in particular upon minimizing RAW hazards which are seen to be the most detrimental type of hazard for the architecture.

The cycle time of the DC is also seen to have a minimal affect on the processor's speed, since other operations can still be executing and initiating during the longer stalls of memory accessing. Furthermore, since arithmetic operations are dominant, the dependencies in the memory unit are less frequent so that a longer cycle time is less detrimental.

8.10 Comparisons

Table 8.3 provides the Mips performance of the *ECSCCESS* processor together with those of other asynchronous superscalar microprocessors previously developed, and for comparison the performance of *ECSTAC* is also quoted (although this has been implemented in CMOS whereas the others have only been simulated in VHDL). The Fred architecture [Ric96] was simulated with exceptionally low gate delays (0.1ns per gate, with a 32 bit adder for

example assumed to take merely 0.5ns), and has therefore been scaled down by a factor of 10 to give an equivalent gate delay to that used in *ECSCCESS*. The quoted gate delay from the SCALP processor [End95b] (for all instructions with hazard minimization) is 48.3ns if a 1ns gate delay is again assumed.

Processor	Fred	SCALP	ECSCCESS	ECSTAC-32
Design style	2P SI	4P SI	2P BD	2P BD
Speed (Mips)	29.8	20.7	141.7	50

Table 8.3: Speed comparisons of various superscalar asynchronous microprocessors.

The *ECSCCESS* processor gives by far the best Mips performance of the superscalar structures, being approximately 5 and 7 times faster than the Fred and SCALP architectures respectively. Both of these processors suffer from the slower SI environment used for their control structures and the need for frequent utilization of the register bank (the Fred architecture has a global register bank whereas the SCALP architecture requires the local register bank to be frequently used).

The performance of *ECSCCESS* is almost 3 times faster than *ECSTAC* when scaled to a 32 bit data path, which seems to imply that a superscalar approach to asynchronous microprocessors is in fact preferable to a pipelined approach. Given that the trend for synchronous microprocessors has also lead to superscalar architectures it is not surprising that in the asynchronous domain a similar trend seems necessary.

8.11 Extensions and improvements

The *ECSCCESS* processor has in fact been designed with support for the more complex interactions required for enabling interrupts, and a general description of the extensions required for this, as well as ideas on how exceptions can be handled, are presented hereafter. Other improvements to the architecture are also discussed.

8.11.1 Incorporating interrupts

A block diagram of the processes which have to be executed during an interrupt is shown in Fig.8.14. The most important consideration is to implement the detection of an interrupt in such a way as to minimize its impact upon the usual cycle time of the processor (when

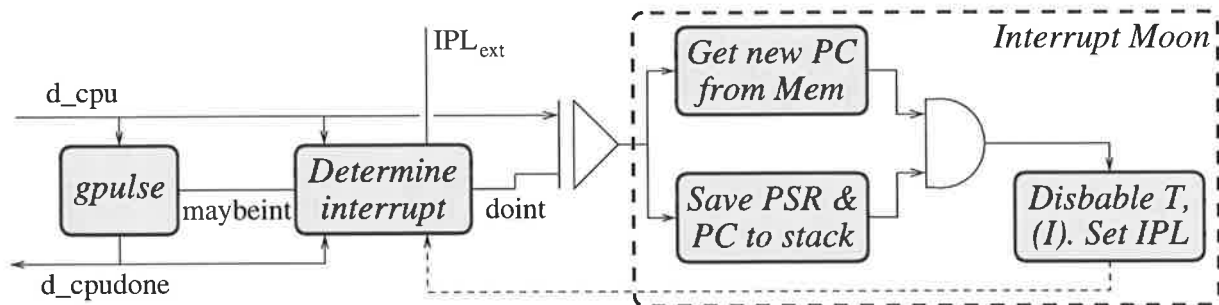


Figure 8.14: Block diagram for processing interrupts.

no interrupts are occurring). To effect this a scheme similar to that of Section 4.4.1 is implemented in which the interrupt priority level of the interrupting process (IPL_{ext}) is latched at the conclusion of one processor cycle and analyzed on the next. In fact the result of the comparison between IPL_{ext} and the current IPL of the PSR is actually latched (being usually high, and low only when an interrupt of high enough priority is detected), producing *maybeint* at the output of the metastability resolver.

This signal is then *and*'ed into the complex gate of Fig.8.9 which governs the production of $\partial cpudone$, and as such the detection of an interrupt (or more specifically, the non-occurrence of one) has essentially *no* impact upon the processor's typical cycle time. Note however that *doint* doesn't actually go high until the latched comparison signal actually indicates a logic one ($> 4V$) whereas *maybeint* may still go low if this signal is metastable (which is detected before the next ∂cpu event by virtue of the IC cycle time). The *send* gate governed by *doint*+*maybeint* is used to halt ∂cpu prior to the *feed* gate if the latched signal is in fact metastable.

If *doint* = 1 then the interrupt processing moon is activated. This initiates a fetch from the address $IVR + IPL_{ext}$ onto a unique bus (*bus2*) from the memory unit whose result gives the address of the relevant interrupt routine to be executed and is multiplexed onto the input to the PC latch of Fig.8.9. Concurrently, the stack moon is activated twice to store the PSR (which contains the current value of the IPL field) and the PC, since this instruction will need to be restarted after a RETI is executed. This process increases the delay of the processor for a CALL instruction by just t_{merge} .

Once these two concurrent operations have completed, the trace mode is disabled by masking its flag in the PSR and then the new IPL is set to IPL_{ext} (note also that interrupts are disabled if this new $IPL=7$). Once this is loaded into the PSR the *maybeint* signal

will go high again, which then enables the new PC location for the interrupt routine to be latched as well as generating $\partial cpudone$. Note that it is the responsibility of the interrupt routine(s) to save the current state of the FU buses into memory or the local register bank (assuming a separate window is used for interrupt storage).

8.11.2 Exception handling

Exceptions can be handled in *ECSCCESS* by assigning a tag to each instruction which is issued into the globe. Furthermore, each instruction which can potentially trap has its tag sent to the exception queue (EQ) which maintains the correct order of exception handling in the event of an out-of order initiation of exceptions. Once an instruction which could trap completes in a FU it sends an event back to the end of the EQ which is stalled until the corresponding instruction tag is present at the output (a similar technique to the order unit of *ECSTAC*). If the FU signals no exception, then no further processing occurs, and the next tag from the EQ is propagated to the output. Otherwise, an exception moon is initiated, and the contents of the EQ nullified to prevent subsequent exceptions until this one has completed. Note that to prevent stalling of the FUs from a stall at the EQ, a 3 stage FIFO per FU is used as a buffer. The exception moon incorporates its detection into the usual IC fetching cycle in the same way as the interrupt moon (with preceding arbitration between these units), and therefore also has a negligible effect on the processor's usual cycle time.

Each FU also maintains a history buffer of its outputs (and internal registers) in a small cyclic RAM structure whose size is governed by the maximum number of instructions which can be issued to it during the longest possible delay for an exception to occur for a preceding instruction (otherwise creating a processor stall). The sun also maintains a similar history of all of its status registers. When an exception routine is entered, each FU is initiated in turn with an EXC instruction which contains the tag for the initially trapped instruction. The result in its history buffer whose tag is nearest but not greater than the exception tag is placed onto the output bus (checked simultaneously across the RAM with the latest entry having precedence), and similarly for the internal registers. Concurrently, the SP register is reloaded with the value corresponding to its exception tag (which is unlikely to have altered), and the corresponding PC and PSR are saved to the

stack so that a RETI instruction from the exception routine can begin again at the point of exception. The address of the exception routine is fetched from the memory unit (prior to having an EXC instruction issued to it) at location $IVR+ETYPE+8$, where $ETYPE$ is a specific identifier for the type of exception which occurred. This address is then loaded into the PC, and once this process and that of issuing EXC instructions to all of the FUs have completed, the gate signal for generating $\partial cpudone$ is sent high which enables the exception routine to be entered.

8.11.3 Reducing the ocean width for WAR and RAW hazards

With the architecture as presented the globe consists of 68 signals per FU, of which 32 are used for controlling WAR hazards. This can be reduced significantly by moving the shore control which implements this into a separate WAR unit.

When an operation to FU_i occurs which sources FU_j , the counter $Cwarin_{i,j}$ is incremented which therefore sets the signal $nostall_j$ low by comparing $Cwarin_{i,j}$ against $Cwarout_{i,j}$ for all i FUs. This functionality is currently contained within FU_j , but is now transferred to the WAR unit which supplies the $nostall$ signal to each FU. Furthermore, the source FU value of “j” is stored in a 3-stage queue (corresponding to the number of FIFO stages in each FU) within the WAR unit for FU_i (a separate queue exists for each FU).

Once FU_i has sourced its data from FU_j , it sends back a single event ∂src_i to the WAR unit (instead of sending 36 events to the ocean as per the current architecture), from which a pulse is generated. The output of the queue will specify the source (FU_j) which will demultiplex this pulse to the counter $Cwarout_{i,j}$, and subsequently set $nostall$ high.

It is evident that the only additional circuitry needed is a 3-stage queue for each FU, however the area of the globe has been reduced from 36 wires per FU to just two (for $nostall$ and ∂src). Also, each FU receives just the one input signal for a WAR hazard as opposed to 36 in the current architecture. Consequently, the ocean width is reduced from 6.2mm to 3.5mm, however the delay for the $nostall$ signals within the globe will be increased slightly by the wire delay between the WAR unit and the relevant FU.

A similar principle can be applied to RAW hazards. The control within the shore of

each FU for managing these hazards can be moved into a separate RAW unit which in fact requires no additional circuitry. This configuration leads to a further reduction in the globe width for RAW hazards from 4 wires per FU to 3 (for *rawokay1, 2* and *dfudone2*), and a reduction in the number of signals multiplexed into each FU from 4 to 2. This is however only a minor improvement in area at the cost of a marginal increase in processing delay (due again to an increase in wire lengths), which may not be worth implementing in practice.

8.12 Summary

The superscalar architecture of *ECSCCESS* has demonstrated a significant speed improvement over the pipelined architecture of *ECSTAC*, which is due to a number of factors including greater parallelism in FU operation, removal of pipeline bottlenecks, distributed hazard control (no global register bank), and global results forwarding. This indicates that as in the synchronous realm a superscalar approach to microprocessor design can yield a higher Mips performance than a pipelined approach. The issue of whether or not an asynchronous implementation can outperform a corresponding synchronous one remains to be answered, and requires identical ISAs and architectures to be implemented.

It should also be noted that *ECSCCESS* has outperformed the other superscalar asynchronous microprocessors by a significant factor in simulation. This can be attributed primarily to the 2P ECS design paradigm which has been shown to result in significant speed improvements over SI and DI implementations.

Chapter 9

Conclusions

THE intended focus of this thesis has been to devise fast asynchronous circuit techniques, since those currently in use suffer from slow handshaking control and excessive circuit complexity. To achieve this aim a more flexible 2P bounded delay design paradigm has been adopted, since the SI and DI paradigms of popular use do not enable sufficiently fast structures to be devised nor enough flexibility in their implementation.

In devising these asynchronous circuits an engineered approach has been taken, since the automatic synthesis from high level specifications does not allow for enough control to be exercised over the low-level gate implementations. Such an approach however requires a knowledge of the optimization techniques and the useful sub-circuits which are available to the designer to enhance the usability of the ECS approach. Therefore this thesis has also focussed on identifying these techniques and explaining their operation and purpose for general use.

As has been shown by the myriad control structures discussed throughout this thesis, the popular belief that 4P circuits are faster than 2P is not necessarily true. Although this has been demonstrated for SI both here and elsewhere, the same cannot be said of BD systems. The 2P ECS circuits presented here have shown an improvement upon the 4P designs in almost every instance, in some cases being up to 4 or 5 times faster.

One reason for this is the excessive control circuits used in speed independent designs. By removing the numerous unnecessary acknowledgements present in this model, the gate count for the 2P bounded delay environment can be significantly reduced. The elimination of these acknowledgements is based on an estimate of comparative gate delays, which is a completely reasonable assumption in all but the rarest of instances as evidenced by the

plethora of working synchronous silicon present in the current market.

Another factor responsible for this speed improvement is the fundamental gate structures which arise from the ECS representation of 2P signalling (which have been shown to be a primitive set of the micropipeline library, among others). By viewing the design process from the basic functionality of these gates, certain control structures can be implemented faster than would otherwise be possible (such as in the splitting of a tree of *select* gates into a row of *feed* gates).

As has also been demonstrated by the numerous control structures presented, from a few gates to a few thousand, the power consumption of the ECS circuits has been very low despite having focussed primarily on high speed implementations, a goal which is often in conflict with the pursuit of low power consumption. Surprisingly, in the majority of cases the power consumption for the faster ECS circuits has in fact been less than those of the 4P SI approaches, whose simpler gate structures are also intended to reduce their power dissipation over 2P designs. This fact can be attributed to the significant reduction in gate counts for the ECS designs which in turn reduce the number of switching transitions (dynamic power dissipation). Therefore focussing on high speed designs has also reduced power consumption, despite first thoughts indicating the contrary.

The shift to a 2P BD model in ECS has enabled pipeline structures to be developed which improve upon the speed performance and in many instances the power dissipation of others which have been previously reported. This is an important issue since pipelines are a fundamental component of many practical applications. Furthermore, since this is the crux of the reasoning behind the hypothesis that synchronous systems should always be faster than asynchronous, the fact that ECS has reduced these handshaking delays now brings this hypothesis into question. Indeed, it would be difficult to clock a $0.7\mu\text{m}$ CMOS FIFO as fast as the state pipeline can cycle (at 360MHz) - even ignoring the issues of skew, routing, and driver sizes (and hence power consumption).

A new approach to self-timing has also been presented in the form of pseudo self-timed circuits. These structures have been shown (for an adder) to be slightly faster than their ST counterparts as well as occupying less area. The robustness of PST designs is expected to be comparable to their ST equivalents since the matched validity path would invariably be implemented in VLSI adjacent to the computational path, and so any process variations are unlikely to cause erroneous behaviour. For some circuits however (such as

an incrementer) the PST approach is not suitable, although dedicated ST implementations of these systems have been shown to be of benefit in reducing the typical computation latency as well as the power dissipation over non-ST structures.

Interestingly, significantly faster synchronous circuits for multiplication than the asynchronous ST implementation presented in this thesis have been reported, although the speed of the latter is still comparable. Note that for larger data widths, the ST structures may become faster than can be implemented synchronously due to their logarithmic dependence on bit width. The ST structures are also smaller and consume less power than their synchronous counterparts since the latter employ more redundancy in their efforts to speed up the worst case computation time.

As an investigation into how ECS can be applied to the design of larger systems two microprocessors have been developed. The first processor *ECSTAC* was implemented and fabricated in the ES2 technology. Its performance was limited by the architecture (and not the ECS paradigm) which utilized an 8 bit data path and a mismatched 24 bit address path. Nonetheless the implementation of this basic pipelined processor is still comparable to others which have been developed.

When scaled to a matched 32 bit data and address path, *ECSTAC* is in fact expected to outperform all of the other 32 bit CMOS processors thus far reported in both 4P and 2P environments (however this comparison is heavily dependent upon the scaling assumptions). The processor has also exhibited a very low power dissipation, and in fact improves on most other processors which have either been explicitly engineered for low power or have been expected to achieve this through the robustness of their design paradigm (by removing all power consuming glitches). This further supports the notion that the ECS structures developed in this thesis are not only fast but are still low in power consumption.

The *ECSTAC* processor also enabled some basic issues of verification to be addressed. In particular, the use of a separate power bus for the delay modelled elements enables a form of frequency tweaking as used in synchronous designs to rectify an incorrect chip. This also reduces the required safety margins for these elements which is often a contributing factor to an increase in pipeline cycle times.

The second processor *ECSCCESS* was designed to better utilize the ST property of

asynchronous systems through superscalar as opposed to pipelined operation. The performance of *ECSCCESS* was simulated to be significantly faster than any other asynchronous superscalar processor previously reported. This is due in part to the ST operation of the FUs and critical computational blocks (such as incrementing the PC), but is also due to the fast ECS control structures used in its implementation. Given that this architecture has resulted in superior performance to other pipelined processors as well, this indicates that future asynchronous microprocessors should (as in the synchronous realm) be based on superscalar operation.

Finally then it is concluded that by using the techniques and fast circuits developed in this thesis, high speed architectures can be developed which reduce (and perhaps even reverse) the performance deficit between asynchronous and synchronous designs. Surprisingly, the asynchronous structures presented here are also low in both area overhead and power dissipation. Although not originally anticipated, these results provide further reasons for implementing ECS circuits in favour of SI or DI equivalents. The most noticeable disadvantage of ECS in comparison to these paradigms is the longer design time to produce reliable circuits. Consequently, it is the authors opinion that automated SI and DI techniques should be applied to ASIC designs in which a fast turn-around time meeting low performance specifications are required, whereas ECS techniques should be applied to the design of custom, high performance chips.

9.1 Further work

Implementing a full 32 bit version of *ECSTAC* would be useful to verify the scaling assumptions made in Section 7.6.5 and to more accurately gauge its performance against other 32 bit CMOS microprocessors. It would also be beneficial to have the *ECSTAC* chip fully tested so that the resilience to operating and process variations of the ECS circuits could be measured, as well as the usefulness and usability of the testability techniques implemented for the processor.

Similarly, implementing *ECSCCESS* in VLSI will enable the assumptions on which its simulations were based to be tested, and a more accurate measurement of its area and processing speed to be determined. This would then enable a more accurate comparison of the superscalar architecture against the pipelined. This implementation of *ECSCCESS*

should also be extended to include interrupt and exception processing which would improve the viability of the architecture for commercial applications. Other extensions such as multiple instruction issue, FU renaming (akin to register renaming), and branch prediction should also be investigated and incorporated into the architecture.

Furthermore it would be useful to have a compiler (or post-processor to a current compiler) which enabled working instruction streams to be tested on the two ECS processors. This would again provide for a better performance comparison.

It would also be worthwhile implementing the speed improvements to the PST multiplier discussed in Section 6.6.7, as well as investigating other multiplier algorithms for their ability to better utilize self-timing. This also applies to the PST adder structure since only the ripple carry approach has been self-timed, whereas it may be that by combining ST and CLA techniques an even faster adder could be devised.

A software package should also be developed which enables the error checking and analysis properties inherent in the ECS framework to be utilized. Such a tool would enable the designer to specify a circuit in a TS format (with hierarchy) with IO constraints which have to be met by the environment, as well as providing a library of useful sub-systems. These can then be checked in the translation from the TS to the chosen simulation suite, and errors in the event control (or the data path) flagged with commentary back to the designer. This would then reduce the design time for developing reliable circuits and improve its usability for new designers.

Appendix A

Fundamental Temporal Equations and Corresponding ECS Gates

Table A.1 provides a list of the fundamental temporal equations together with their corresponding ECS gates and circuit representations.

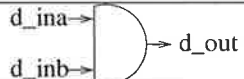

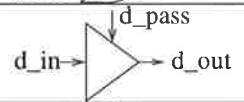
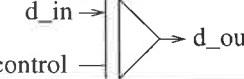
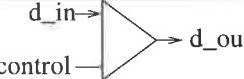

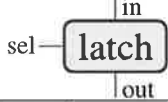

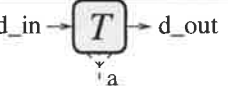
TE	ECS Gate	Circuit Symbol
$\partial out \leftarrow \partial ina \cdot \partial inb$	<i>cgate</i>	
$\partial out \leftarrow \partial ina + \partial inb$	<i>merge</i>	
$\partial out \leftarrow \partial pass > \partial in$	<i>restore</i>	
$\partial out \leftarrow \partial in > control$	<i>feed</i>	
$\partial out \leftarrow \partial in \cdot control$	<i>send</i>	
$out \leftarrow \partial ina \cup \partial inb$	<i>xor</i>	
$out \leftarrow in > sel$	<i>latch</i>	
$c \leftarrow \mathcal{F}(a, b, \dots)$	<i>logic</i>	
$\partial out \leftarrow a : b : \dots : \partial in$	<i>delay</i>	

Table A.1: Fundamental temporal equations and their corresponding gates.

The following are examples of TE specifications which are invalid. This may be due to an invalid operation between data signals and event lines, an assignment of a data signal to an event line, or an input event occurring without a corresponding output event (or vice versa).

$$\begin{array}{ll}
 c \leftarrow \partial x > \dots & \partial x \leftarrow c > \dots \\
 \partial z \leftarrow \partial x + c & c \leftarrow \partial x + b \\
 c \leftarrow x.\partial z & c \leftarrow \partial a.\partial b \\
 c \leftarrow \partial a + \partial b & \partial z \leftarrow a + b \\
 a \leftarrow b > \partial z & \partial z \leftarrow a U b
 \end{array}$$

Appendix B

ISA of the *ECSTAC* Microprocessor

There are a total of 47 distinct instruction types excluding mode variants, and 87 including mode variants. 16 registers are available for use and there are also dedicated registers for the program counter, stack pointer, and flags register (FR). The FR is configured as:

DC	IC	-	zero	parity odd	sign	overflow	carry
----	----	---	------	------------	------	----------	-------

where DC and IC represent the state (active or bypassed) of the data and instruction caches respectively.

B.1 Memory instructions

The 16 register banks can be partitioned into four register quadruples for data movement instructions, denoted simply as Q0, Q4, Q8, and Q12, with the lowest numbered register containing the lowest byte of the 24 bit address.

B.1.1 Two byte instructions

2	2	4	4	4
op	mode	Qx	Rz	Qy

op=00 for the LD (load) instruction, and op=01 for the ST (store) instruction. Rz represents the register location to load data into or store data from, and mode determines the interpretation of Qx and Qy as follows:

- **mode=00 (register mode):** Qx and Qy represent register quadruples, and the actual memory address is computed as the sum of the two 24 bit values stored in these registers.

- **mode=01 (offset mode):** Qx represents a 24 bit address as above, but Qy represents an unsigned four bit constant, which is added to the register address to give the actual memory address.

B.1.2 Four byte instruction

2	2	4	8	8	8
op	mode	Rz	MemLow	MemMid	MemHig

op and Rz are as given for the two byte instructions, and **mode=10 (direct mode)**. The actual memory address is given by (MemHig || MemMid || MemLow).

B.1.3 The unused mode

mode=11 is not used for any data movement operations. It is used instead for the JUMP instruction (see Section B.3.2).

B.2 ALU instructions

B.2.1 Two byte instruction (short mode)

2	2	4	4	4
op	mode	code	Rz	Ry

op=10 indicates an ALU operation, and code indicates the specific ALU operation as given in Table B.1. mode determines the interpretation of Rz and Ry as follows:

- **mode=00 (register mode):** Rz represents the destination register for the result, as well as the source register of the first argument. Ry represents the source register of the second argument. For those functions with only one argument, this is taken from Ry, thereby enabling maximum flexibility of register transfers. Single argument functions will always use this mode.
- **mode=01 (offset mode):** Rz represents the destination register for the result, as well as the source register of the first argument. Ry represents an unsigned 4 bit constant value as the second argument.

Code	Instruction	Arguments	Meaning
0000	NEG	1	Arithmetic sign negation
0001	MOVE	1	Move data between registers
0010	INC	1	Increment by one
0011	DEC	1	Decrement by one
0100	SHLL	1	Shift left logical (wrap around bits)
0101	SHLA	1	Shift left arithmetic (pack with 0's)
0110	SHRL	1	Shift right logical
0111	SHRA	1	Shift right arithmetic
1000	SUBU	2	Unsigned subtraction
1001	SUBS	2	Signed subtraction
1010	ADDU	2	Unsigned addition
1011	ADDS	2	Signed addition
1100	NOT	1	Logical bit negation
1101	AND	2	And function
1110	OR	2	Or function
1111	XOR	2	Exclusive-or function

Table B.1: ALU instructions.

B.2.2 Three byte instructions (long mode)

2	2	4	4	4	4	4
op	mode	Rx	Rz	Ry	offhi	code

op and code are as given for the two byte instructions, and mode determines the interpretation of Rz, Rx and Ry as follows:

- **mode=10 (register mode):** Rz represents the destination register for the result, Rx represents the source register of the first argument, and Ry represent the source register of the second argument. offhi is unused.
- **mode=11 (offset mode):** Rz represents the destination register for the result, Rx represents the source register of the first argument, and (offhi || Ry) represents an 8 bit constant value as the second argument.

B.3 Branch instructions

B.3.1 One byte instruction - CALL

4	4
op	Qz

where op=1111 for the CALL instruction, and Qz refers to the register quadruple containing the memory address to CALL to.

Code	Instruction	Meaning
0000	JUMP	Unconditional jump
0010	JMPZ	Jump when <i>result</i> = 0 (zero bit)
0011	JMPX	Jump when <i>result</i> ≠ 0 (inverse zero bit)
0100	JMPN	Jump when <i>result</i> < 0 (sign bit)
0101	JMPG	Jump when <i>result</i> ≥ 0 (inverse sign bit)
0110	JMPL	Jump when <i>result</i> ≤ 0 (sign OR zero)
0111	JMPP	Jump when <i>result</i> > 0 (sign NOR zero)
1000	JMPV	Jump when parity bit is set
1001	JPNV	Jump when parity bit is not set
1010	JMPO	Jump when overflow bit is set
1011	JPNO	Jump when overflow bit is not set
1100	JMPC	Jump when carry bit is set
1101	JPNC	Jump when carry bit is not set
1110	JPOC	Jump when carry or overflow bits are set
1111	JNOC	Jump when neither carry nor overflow bits are set

Table B.2: Branch instructions.

B.3.2 Two byte instructions - BRANCH

4	4	4	4
op/mode	code	Qz	offlo

where $op=0(mode)11$ represents the conditional or unconditional jump instruction (where the condition being tested is given by the contents of the FR), code represents the type of comparison to be made (if any) on the contents of the FR as given in Table B.2, and mode determines the interpretation of Qz and offlo as follows:

- **mode=0 (offset mode):** (Qz || offlo) represents a *signed* 8 bit offset address to be branched to relative to the current PC address.
- **mode=1 (register mode):** Qz represents the register quadruple containing the address to be jumped to and offlo is unused.

B.4 Stack instructions

4	4
op	Rx

$op=1101$ represents a PUSH operation, and $op=1100$ represents a POP. Rx represents the register whose data is to be pushed, or to where data will be popped. Note that more stack instructions are specified in the following section.

B.5 Special instructions

4	4
op	code

op=1110 represents a special instruction to be interpreted by Table B.3.

Code	Instruction	Meaning
0000	NOOP	Do Nothing
0001	RETN	Returns from a call instruction
0010	FLSH	Flushes the contents of the data cache
0100	ICDS	Instruction cache disable
0101	ICEN	Instruction cache enable
0110	DCDS	Data cache disable
0111	DCEN	Data cache enable
1000	HALT	Halts processor after 4 successive instructions
1100	POPF	Pops the contents of the stack into the FR
1101	PSHF	Pushes the contents of the FR onto the stack
1111	TRSP	Transfers Q12 to the stack pointer

Table B.3: Special instructions.

Appendix C

ISA of the *ECSCCESS* microprocessor

The *ECSCCESS* ISA provides for 16 basic instruction types (encoded in the *op* field below) of which 13 have currently been defined. Each of these instruction types are encoded in one of the following four formats:

Bits	31→28	27	26→23	22	21→19	18→12	11→7	6	5→1	0
Btype	op	ms	bramt (low 27 bits)							
Ltype	op	dest		code		constant				
Ftype	op	dest		code		immhi	src1	u1	src2	u2
Rtype	op	dest		wr	winreg				src2	u2

Table C.1: Instruction formats.

The *op* field specifies the particular type of instruction which is being executed, as shown in the table below. Only for *op*'s 8 through 11 is the globe activated.

op	instruction	op	instruction	op	instruction	op	instruction
0	JUMP offset	1	JCOND offset	2	CALL offset	3	RETC
4	JUMP bus0	5	LDSP	6	CALL bus0	7	JCOND bus0
8	FU bus	9	FU imm	10	MOVE	11	LDC
12	-	13	-	14	-	15	INT

Table C.2: Fundamental instruction set.

C.1 Branch instructions

The branch instructions comprise *op* fields 0 through 7, and are all encoded as a Btype. The JUMP instruction performs an unconditional jump to a new PC location, whereas the JCOND instruction performs this only if the MSB of *bus1* is high (which should be from a comparator unit tested prior to the conditional jump). A CALL instruction branches to

a subroutine, and a RETC instruction returns from it (the *ms* and *bramt* fields for this instruction are irrelevant).

For the branch-bus0 instructions, the *ms* and *bramt* fields are irrelevant, and the branch offset is given by the value of *bus0* which is added to the current PC value to give the branch target. For the branch-offset instructions, the *ms* and *bramt* fields collectively represent a 28 bit offset which is sign extended to provide the offset for the branch target addition.

A LDSP instruction loads the stack with the 32 bit quantity $ms + bramt + 1111$, which represents the location in memory of the decrementing stack to be used if the on-chip stack fills up. Note that *ECSCCESS* implements a 32 register on-chip stack (although any depth can be implemented in practice), so that the memory stack will only be needed for code streams with more than 32 embedded levels of subroutines (although an interrupt uses 2 stack registers as well). As such the off-chip memory will rarely ever be used.

C.2 Interrupt instructions

These are also encoded as a Btype. If $ms = 0$ then an IVRL instruction occurs, which loads the interrupt vector register (IVR) with $bramt + 00000$, otherwise the upper 5 bits of the *bramt* field are used to indicate one of the following instructions:

bramt (26 → 22)	instruction	meaning
0 - - - -	RETI	Return from interrupt
1 0 1 1 -	EI	Enable interrupts
1 0 1 0 -	DI	Disable interrupts
1 0 0 1 -	ET	Enable trace mode
1 0 0 0 -	DT	Disable trace mode
1 1 I_2 I_1 I_0	SETI	Set interrupt priority level to I

Table C.3: Interrupt instructions.

Interrupts can be enabled or disabled with the EI and DI commands, and the conclusion of an interrupt processing routine is indicated with a RETI instruction. A 7 level interrupt priority level (IPL) is facilitated, with the highest level (7) being non-interruptible. When an interrupt occurs, its IPL is compared against that of the processor, and if greater then the interrupt routine located at the address pointed to by $IVR + IPL$ in memory is executed.

A trace mode is also facilitated by the ISA of *ECSCCESS*. If enabled, then at the conclusion of each instruction the trace routine located at the address pointed to by IVR in memory is executed. This can be useful for customizing the analysis and debugging of instruction streams.

C.3 MOVE instruction

The MOVE instruction is analogous to register moves in most other processors, but in this instance it transfers the bus value from one FU to the bus value of another (bypassing the FUs operation). It is encoded as an Ftype with the *code*, *immhi*, and *src1* fields being irrelevant. The *dest* field indicates the destination FU for the transfer, and the *src2* field indicates the source FU. For this operation, $u1 = 0$ and $u2 = 1$, implying that only one source FU (*src2*) is needed for the instruction.

The register bank provides a slight exception to this. If the *wr* field is high, then the bus value of *src2* is not only transferred to *dest*, but is also written to the register location specified by the 16 bit signal *winreg*. If low, then the value is simply transferred as with any other FU. Note that the register unit therefore uses the Rtype of encoding (which is very similar to the Ftype).

C.4 LDC instruction

This instruction utilizes the Ltype instruction format and is used to load a constant value (specified in the *constant* field) or a status register onto the bus of the FU specified in *dest*. How the *constant* value is interpreted is given by *code* in Table C.4.

code	32 bit constant to load (<i>const32</i>)
0 0 0 -	$const32_{31 \rightarrow 19} + constant$
0 0 1 -	sign extended <i>constant</i>
0 1 0 -	$constant * 2^{13} + const32_{12 \rightarrow 0}$
0 1 1 -	$constant * 2^{13}$ (left shift 13 bits)
1 0 0 -	current PC
1 0 1 -	SP (for external memory stack)
1 1 0 -	IVR
1 1 1 -	PSR

Table C.4: 32 bit constants for the LDC instruction.

The first four codes enable the 19 bit *constant* field to be loaded into the upper or lower portion of the 32 bit *const32* (which goes to the FU). The other portion is either left unchanged (enabling a full 32 bit value to be loaded in two instructions) or is sign extended for a low portion load (enabling a 19 bit signed constant to be loaded with one instruction) or set to zero (for a high portion load).

The latter four codes enable the process registers of *ECSCCESS* to be loaded into the globe for manipulation. The PSR (process status register) contains the current state of the interrupt and trace processes in the format below:

Bit(s)	PSR entry	Meaning
31	TS	Trace status (on or off)
30	IS	Interrupt status
29 → 27	IPL	Current interrupt priority level
26 → 0	-	Unused

C.5 FU instructions

All FUs in the processor are triggered by one of two FU instructions in the ISA. The FUbus instruction encodes two source FUs for the operation in *src1* and *src2* and the destination FU in *dest*, whose actual functionality is decoded therein by the *code* field. Instructions which use less than 2 operands will have the relevant *u1* and/or *u2* fields set low, indicating that the encoded source FU is not required. Note that these instructions could also utilize the *immhi* and *src1* fields (and *src2* for instructions with no operands) to extend the code field for the operation. The FUimm instruction encodes a sign extended 13 bit value of *immhi* + *src1* + *u1* in place of the *src1* FU used in the FUbus mode.

Although the architecture of *ECSCCESS* enables up to 32 FUs of any type to be employed, clearly these must be specified explicitly for any given implementation. A proposed allocation is given in Table C.5, and the specific set of instructions executed within each FU (given by the *code* field in general) are discussed in the following sections.

C.5.1 Register unit (reg)

The register unit is the only FU whose instructions are encoded as the Rtype. The *src2* field specifies a source FU for a write operation (when *wr* = 1), and for a read operation (when *wr* = 0) this field is irrelevant (the *u2* field will therefore be set low for a read operation). The 16 bit *winreg* field specifies the register to which (or from which)

dest	FU	dest	FU	dest	FU	dest	FU
0	reg0	1	reg1	2	reg2	3	reg3
4	aid0	5	aid1	6	aid2	7	aid3
8	mds0-b0	9	mds0-b1	10	mds1-b0	11	mds1-b1
12	cmp	13	shl	14	icop	15	mem
16	freg0-b0	17	freg0-b1	18	freg1-b0	19	freg1-b1
20	faid0-b0	21	faid0-b1	22	faid1-b0	23	faid1-b1
24	fmds0-b0	25	fmds0-b1	26	fmds1-b0	27	fmds1-b1
28	fcmp-b0	29	fcmp-b1	30	fcop-b0	31	fcop-b1

Table C.5: Proposed FU allocations for *ECSCCESS*.

the source FU's data will be written (or read). Clearly $2^{16} = 65536$ registers is a lot, therefore it could be expected that the registers will be partitioned into windows (giving a hierarchical register structure), perhaps reserving one or more windows for storage and retrieval during an interrupt or trace routine. Furthermore, it is possible to encode within *winreg* a field which enables up to 4 multiple register reads, and another to implement indexing as in the memory unit discussed in Section C.5.6.

The *dest* field contains the FU location of the register unit, and in the proposed FU allocations this occupies units 0 to 3. Although only one actual register unit is present, these four output buses enable a read to place its result onto a specific bus, enabling previously read data to be retained if necessary.

C.5.2 Arithmetic unit (aid)

The AID performs basic arithmetic operations as given by Table C.6, and four separate AIDs are provided in the FU allocation to enhance parallelism. Note that *arg2* and *arg1* come from the *src2* and *src1* source FUs respectively (although the latter may in fact be encoded as a 13 bit signed constant), and that an internal FR is used to retain flags information which can be read out with a FLAG instruction. The structure of the FR is shown in Section C.5.5 (minus the branch bit).

C.5.3 Multiply, divide, and sqrt unit (mds)

This unit implements signed and unsigned multiply, divide, and square root functions according to Table C.7.

Code	Instruction	Meaning
0	ADD	$arg2 + arg1$
4	SUB	$arg2 - arg1$
8	INC	increment $arg2$
10	NEG	negate (two's complement) $arg2$
11	DEC	decrement $arg2$
15	FLAG	output the FR

Table C.6: Arithmetic instructions.

Code	Instruction	Meaning
0, 1	MULU, MULS	Unsigned and signed multiplication
2, 3	DIVU, DIVS	Unsigned and signed division
4	SQRT	Unsigned square root of $arg2$

Table C.7: Multiply, divide, and sqrt instructions.

A 32 bit multiplication can result in a 64 bit product, therefore to enable the full result to be provided in one operation each MDS unit (of which there are two specified) is spread across two buses (as indicated by the -b0 and -b1 postfixes). The division and sqrt functions similarly require two buses, providing the quotient onto b1 and the remainder onto b0.

C.5.4 Shifter and logical unit (shl)

This simple unit performs arithmetic, logical, and wrapped shifts of $arg2$ (with the $src1$ field specifying the number of bits to shift by) as well as performing logical functions as per the following table.

Code	0	1	2	4	5	6
Instruction	SLA	SLL	SLW	SRA	SRL	SRW

Code	8	9	10	11	12	13	14
Instruction	AND	NAND	OR	NOR	XOR	XNOR	NOT ($arg2$)

Table C.8: Logical and shifting instructions.

C.5.5 Comparator (cmp)

The comparator unit enables three basic types of operations. The CMP2 instruction compares two signed source values and sets the branch bit (the MSB of the output bus) according to the type of comparison required. This bit can also be set according to the parity of $arg2$.

The ASC instruction performs addition and subtraction of the two source values and places the result onto the output bus as well as computing flags based on the result. Furthermore, the two arguments are also compared as per a CMP2 operation (although whether or not the arguments are signed can be specified explicitly), again setting the relevant flags in the FR. Whenever a CMP2 or CMP0 instruction occurs, the FR is placed onto the output bus together with the branch bit as shown below:

Bit	31	18	17	16	9	8	3	2	1	0
Flag	branch	qc	hc	c	u	v	po	lt	gt	eq

where each flag represents in turn: a carry out of bits 8, 16, and 32; underflow (only relevant for the floating point units described later); overflow; odd parity for the result; $arg2 < arg1$; $arg2 > arg1$; and $arg2 = arg1$. A CMP0 instruction sets the branch bit according to the flags set by the previous ASC instruction. The encoding for each of these comparator instructions is given in Table C.9. Note that the CMP0 instruction utilizes the *src1* field in its decoding, which is acceptable since it has no arguments.

Code		1ab0 (ASC)	1ab1 (CMP0)			0abc (CMP2)
a	b	Operation	<i>src1</i> = 1	<i>src1</i> = 2	<i>src1</i> = 4	Operation
0	0	ADD, signed CMP	eq	v	c	$arg2 = arg1$
0	1	ADD, unsigned CMP	gt	u	hc	$arg2 > arg1$
1	0	SUB, signed CMP	lt	-	qc	$arg2 < arg1$
1	1	SUB, unsigned CMP	-	po	-	$arg2$ parity odd

Table C.9: Comparator instructions.

If the *c* bit for a CMP2 operation is high, then the inverse comparison is performed, and similarly for CMP0 if the MSB of *src1* is set.

C.5.6 Memory unit (mem)

The memory unit performs load and store operations to external memory (perhaps with an intervening DC), and the *code* field is segregated into the LS, UL, ID, and BA bits (from MSB to LSB) which imply the following:

- If LS=1 then perform a LD from address $arg2$, otherwise perform a ST to address $arg2$ with the data from $arg1$.
- If UL=1 then use the last modified address (LMA) instead of the address of $arg2$, otherwise save $arg2$ into the LMA as well as performing the memory operation.

- If ID=1 then increment the LMA, otherwise decrement it.
- If BA=1 then perform the increment or decrement of the LMA before accessing memory, otherwise do it afterwards.

The ID and BA bits are irrelevant if UL=0. Using pre and post incrementing and decrementing of the LMA enables consecutive memory operations (indexing) to be implemented without requiring the arithmetic units.

C.5.7 Floating point units and co-processors

There is also provision in the ISA of *ECSCCESS* for floating point (FP) units, however since these may operate on double and single precision numbers, two output buses are needed for each FU. There are 2 register read buses, 2 Faid units, 2 Fmds units, and a FP comparator, each essentially analogous to their integer counterparts. Since no FP units have been implemented in the current *ECSCCESS* architecture no instruction codes have been specified. A FP and integer co-processor (fcop and icop) can also be used, with field encodings specific to whatever unit is attached.

Bibliography

- [ABV⁺95] A. J. Acosta, M. Bellido, M. Valencia, A. Barriga, R. Jiménez, and J. L. Huertas. New CMOS VLSI linear self-timed architectures. In *Asynchronous Design Methodologies*, pages 14–23. IEEE Computer Society Press, May 1995.
- [AML95a] Sam S. Appleton, Shannon V. Morton, and Michael J. Liebelt. Cache design for an asynchronous VLSI RISC processor. In *Proc. 13th Australian Microelectronics Conference*, pages 91–96, July 1995.
- [AML95b] Sam S. Appleton, Shannon V. Morton, and Michael J. Liebelt. The design of a fast asynchronous microprocessor. *IEEE Technical Committee on Computer Architecture Newsletter*, October 1995.
- [AML96] Sam S. Appleton, Shannon V. Morton, and Michael J. Liebelt. A new technique for high-speed asynchronous pipeline control. *Electronics Letters*, 32(21):1973–1974, October 1996.
- [App96] Sam Appleton. Implementation of instruction and data caches for the EC-STAC microprocessor. Report HPCA-ECS-96/02, Dept. of Electrical and Electronic Engineering, The University of Adelaide, Adelaide, South AUSTRALIA, June 1996.
- [BE89] J. A. Brzozowski and J. C. Ebergen. Recent developments in the design of asynchronous circuits. In J. Csirik, J. Demetrovics, and F. Gécseg, editors, *Fundamentals of Computation Theory, FCT'89*, volume 380 of *Lecture Notes in Computer Science*, pages 78–94, FCT'89, Szeged, Hungary, 1989. Springer-Verlag.
- [Bed62] O.J. Bedrij. Carry-select adder. *IEEE Transactions on Electronic Computers*, EC-11:340–346, 1962.
- [BK82] R.P. Brent and H.T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, C-31(3):260–264, 1982.
- [Boo51] A.D. Booth. A signed binary multiplication technique. *Quart. J. Mech. App. Math.*, 4(2):236–240, 1951.
- [BR95] J. A. Brzozowski and K. Raahemifar. Testing C-elements is not elementary. In *Asynchronous Design Methodologies*, pages 150–159. IEEE Computer Society Press, May 1995.

- [BS89] Erik Brunvand and Robert F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 262–265. IEEE Computer Society Press, November 1989.
- [Chu87a] Tam-Anh Chu. Synthesis of self-timed VLSI circuits from graph-theoretic specifications. In *Proc. International Conf. Computer Design (ICCD)*, pages 220–223. IEEE Computer Society Press, 1987.
- [Chu87b] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, June 1987.
- [CL86] Tam-Anh Chu and Clement K. C. Leung. Design of VLSI asynchronous FIFO queues for packet communication networks. In *Proc. International Conference on Parallel Processing*, pages 397–400, August 1986.
- [CL95] Chih-Ming Chang and Shih-Lien Lu. Design of a static MIMD data flow processor using micropipelines. *IEEE Transactions on VLSI Systems*, 3(3):370–378, September 1995.
- [CM73] T. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22(4):421–422, April 1973.
- [CZ94] J.F. Chappel and S.G. Zaky. A delay-controlled phase-locked loop to reduce timing errors in synchronous/asynchronous communication links. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 156–165, November 1994.
- [DA93] Hema Dhanesha and Alexander Albicki. Self-timed adder with pipelined output. In *Proceedings of the Midwest Symposium on Circuits and Systems*, pages 855–858, 1993.
- [DDH91] Mark E. Dean, David L. Dill, and Mark Horowitz. Self-timed logic using current-sensing completion detection (CSCD). In *Proc. International Conf. Computer Design (ICCD)*, pages 187–191. IEEE Computer Society Press, October 1991.
- [DEC88] Digital Equipment Corporation: DEC. DECChip 21064-AA RISC microprocessor preliminary data sheet. Technical report, D.E.C., Maynard, MA, U.S.A., 1988.
- [Den85] Peter J. Denning. The science of computing: The arbitration problem. *American Scientist*, 73:516–518, December 1985.
- [Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [Dil89] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.
- [DW95] Paul Day and J. Viv Woods. Investigation into micropipeline latch design styles. *IEEE Transactions on VLSI Systems*, 3(2):264–272, June 1995.

- [DWA⁺92] D. Dobberpuhl, R. Witek, R. Allmon, R. Anglin, S. Britton, L. Chao, R. Conrad, D. Denver, B. Gieseke, G. Hoepfner, J. Kowaleski, K. Kuchler, M. Ladd, M. Leary, L. Madden, E. McLellan, D. Meyer, J. Montanaro, D. Priore, V. Rajagopalan, S. Samudrala, and S. Santhanam. A 200 MHz 64b dual-issue CMOS microprocessor. *IEEE Journal of Solid-State Circuits*, 27(11):1555–1565, November 1992.
- [Ebe89] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*, volume 56 of *CWI Tract*. Centre for Mathematics and Computer Science, 1989.
- [EBG93] J. C. Ebergen, P. F. Bertrand, and S. Gingras. Solving a mutual exclusion problem with the RGD arbiter. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 137–147. Elsevier Science Publishers, 1993.
- [ECFS95] C. J. Elston, D. B. Christianson, P. A. Findlay, and G. B. Steven. Hades - towards the design of an asynchronous superscalar processor. In *Asynchronous Design Methodologies*, pages 200–209. IEEE Computer Society Press, May 1995.
- [End95a] Philip B. Endecott. Parallel structures for asynchronous microprocessors. *IEEE Technical Committee on Computer Architecture Newsletter*, October 1995.
- [End95b] Philip B. Endecott. *SCALP: A Superscalar Asynchronous Low-Power Processor*. PhD thesis, Dept. of Computer Science, University of Manchester, U.K., 1995.
- [EP92] Jo C. Ebergen and Ad M. G. Peeters. Modulo-N counters: Design and analysis of delay-insensitive circuits. In Jørgen Staunstrup and Robin Sharp, editors, *Designing Correct Circuits*, volume A-5 of *IFIP Transactions*, pages 27–46. Elsevier Science Publishers, 1992.
- [FDG⁺93] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In T. Yanagawa and P. A. Ivey, editors, *Proceedings of VLSI 93*, pages 5.4.1–5.4.10, September 1993.
- [FDG⁺94] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, S. Temple, and J. V. Woods. The design and evaluation of an asynchronous microprocessor. In *Proc. International Conf. Computer Design (ICCD)*. IEEE Computer Society Press, October 1994.
- [FES94] Craig Farnsworth, Doug Edwards, and Shiv Sikand. Utilizing dynamic logic for low power consumption in asynchronous circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 186–194, November 1994.
- [FL96] S. B. Furber and J. Liu. Dynamic logic in four-phase micropipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

- [Fur96] S. B. Furber. Amulet2e: Invited lecture. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.
- [Gar93] Jim D. Garside. A CMOS VLSI implementation of an asynchronous ALU. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 181–207. Elsevier Science Publishers, 1993.
- [GD85] L.A. Glosser and D.W. Dobberpuhl. *The Design and Analysis of VLSI Circuits*. Addison-Wesley, 1985.
- [GJ90] Ganesh Gopalakrishnan and Prabhat Jain. Some recent asynchronous system design methodologies. Technical Report UUCS-TR-90-016, Dept. of Computer Science, Univ. of Utah, October 1990.
- [GJ95] E. Grass and S. Jones. Asynchronous circuits based on multiple localised current-sensing completion detection. In *Asynchronous Design Methodologies*, pages 170–177. IEEE Computer Society Press, May 1995.
- [Hau93] Scott Hauck. Asynchronous design methodologies: An overview. Technical Report TR 93-05-07, Department of Computer Science and Engineering, University of Washington, Seattle, 1993.
- [Hau95] Scott Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1), January 1995.
- [Haz92] Pieter J. Hazewindus. *Testing Delay-Insensitive Circuits*. PhD thesis, California Institute of Technology, 1992.
- [HBB95] Henrik Hulgaard, Steven M. Burns, and Gaetano Borriello. Testing asynchronous circuits: A survey. *Integration, the VLSI journal*, 19(3):111–131, November 1995.
- [HC95] Calvin J. A. Hsia and C. Y. Roger Chen. Synthesis of asynchronous circuits — testing unique circuit behavior of signal transition graphs. In *Proc. International Symposium on Circuits and Systems*, pages 1074–1077, 1995.
- [HF89] I.S. Hwang and A.L. Fisher. Ultrafast compact 32-bit CMOS adders in multiple-output domino logic. *IEEE Journal of Solid-State Circuits*, 24(2):358–369, April 1989.
- [Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Mateo, California, 1990.
- [JPKJ95] Sung Tae Jung, Uun Sei Park, Junk Sik Kim, and Chu Shik Jhon. Automatic synthesis of gate-level speed-independent control circuits from signal transition graphs. In *Proc. International Symposium on Circuits and Systems*, pages 1411–1414, 1995.

- [JU90a] Mark B. Josephs and Jan Tijmen Udding. An algebra for delay-insensitive circuits. In Robert P. Kurshan and Edmund M. Clarke, editors, *Proc. International Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 343–352. Springer-Verlag, 1990.
- [JU90b] Mark B. Josephs and Jan Tijmen Udding. The design of a delay-insensitive stack. In G. Jones and M. Sheeran, editors, *Designing Correct Circuits*, pages 132–152. Springer-Verlag, 1990.
- [KB95] Ajay Khoche and Erik Brunvand. Testing self-timed circuits using partial scan. In *Asynchronous Design Methodologies*, pages 160–169. IEEE Computer Society Press, May 1995.
- [KdSRA91] S. Karthik, I. de Souza, J. T. Rahmeh, and J. A. Abraham. Interlock schemes for micropipelines: Application to a self-timed rebound sorter. In *Proc. International Conf. Computer Design (ICCD)*, pages 393–396. IEEE Computer Society Press, 1991.
- [Kel74] Robert M. Keller. Towards a theory of universal speed-independent modules. *IEEE Transactions on Computers*, C-23(1):21–33, January 1974.
- [Kes95] Joep Kessels. VLSI programming of a low-power asynchronous Reed-Solomon decoder for the DCC player. In *Asynchronous Design Methodologies*, pages 44–52. IEEE Computer Society Press, May 1995.
- [KKTV92] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. On self-timed behavior verification. In *Proceedings of ACM TAU 92*, March 1992.
- [LCT⁺95] Lavi A. Lev, A. Charnas, M. Tremblay, A. R. Dalal, B. A. Frederick, C. R. Srivatsa, D. Greenhill, D. L. Wendell, D. D. Pham, E. Anderson, H. K. Hingarh, I. Razzack, J. M. Kaku, K. Shin, M. E. Levitt, M. Allen, P. A. Ferolito, R. L. Bartolotti, R. K. Yu, R. J. Melanson, S. I. Shah, S. Nguyen, S. S. Mitra, V. Reddy, V. Ganesan, and W. J. de Lange. A 64-b microprocessor with multimedia support. *IEEE Journal of Solid-State Circuits*, 30(11):1227–1238, November 1995.
- [LZB92] J.J. Laurin, S.G. Zaky, and K.G. Balmain. EMI-induced delays in digital circuits: prediction. In *Proc. IEEE Symp. on Electromagnetic Compatability*, pages 443–448, August 1992.
- [MAL94] Shannon V. Morton, Sam S. Appleton, and Michael J. Liebelt. An event controlled reconfigurable multi-chip FFT. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 144–153, November 1994.
- [MAL95] Shannon V. Morton, Sam S. Appleton, and Michael J. Liebelt. ECSTAC: A fast asynchronous microprocessor. In *Asynchronous Design Methodologies*, pages 180–189. IEEE Computer Society Press, May 1995.
- [Mar86] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

- [Mar90] Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.
- [MBL⁺89a] Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The design of an asynchronous microprocessor. In Charles L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 351–373. MIT Press, 1989.
- [MBL⁺89b] Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The first asynchronous microprocessor: the test results. *Computer Architecture News*, 17(4):95–110, June 1989.
- [McA92] Anthony J. McAuley. Dynamic asynchronous logic for high-speed CMOS systems. *IEEE Journal of Solid-State Circuits*, 27(3):382–388, March 1992.
- [ML93] Shannon V. Morton and Michael J. Liebelt. A 100 Mips event controlled ALU. In *Proc. 12th Australian Microelectronics Conference*, pages 159–164, October 1993.
- [MM82] J.V. McCanny and J.G. McWhirter. Completely iterative, pipelined multiplier array suitable for VLSI. *IEE Proceedings-G*, 129(2):40–46, April 1982.
- [MM93] Chris J. Myers and Teresa H.-Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.
- [MRW96] S. Moore, P. Robinson, and S. Wilcox. Rotary pipeline processors. *Electronics Letters*, 143(5):259–265, September 1996.
- [MU] University Of Manchester: MU. The Asynchronous Logic Home Page. <http://www.cs.man.ac.uk/amulet/async/index.html>. E-mail address: jgarside@cs.man.ac.uk.
- [Nan95] Takashi Nanya. a quasi-delay-insensitive microprocessor: TITAC-I. In *1995 Israel Workshop on Asynchronous VLSI*, pages 95–102. VLSI Systems Research Centre, Technion - Israel Institute of technology, March 1995.
- [Pav94] N. C. Paver. *The Design and Implementation of an Asynchronous Microprocessor*. PhD thesis, Department of Computer Science, University of Manchester, June 1994.
- [PDF⁺92] N. C. Paver, P. Day, S. B. Furber, J. D. Garside, and J. V. Woods. Register locking in an asynchronous microprocessor. In *Proc. International Conf. Computer Design (ICCD)*, pages 351–355. IEEE Computer Society Press, October 1992.
- [Pee] Ad Peeters. The Asynchronous Bibliography. Available for anonymous ftp at <ftp://ftp.win.tue.nl/pub/tex/async.bib.Z>. Corresponding e-mail address: async-bib@win.tue.nl.
- [PG93] R. Puri and J. Gu. Signal transition graph constraints for speed-independent circuit synthesis. In *Proc. International Symposium on Circuits and Systems*, volume 3, pages 1686–1689. IEEE Computer Society Press, 1993.

- [Puc90] Douglas A. Pucknell. *Fundamentals of Digital Logic Design with VLSI Circuit Applications*. Silicon Systems Engineering Series. Prentice-Hall, 1990. Editor: Kamran Eshraghian.
- [RB96] William F. Richardson and Erik Brunvand. Architectural considerations for a self-timed decoupled processor. *Electronics Letters*, 143(5):251–257, September 1996.
- [Ric96] William F. Richardson. *Architectural Consideration in a Self-Timed Processor Design*. PhD thesis, Dept. of Computer Science, University of Utah, U.S.A., February 1996.
- [Ron94] Marly Roncken. Partial scan test for asynchronous circuits illustrated on a DCC error corrector. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 247–256, November 1994.
- [SK93] O. Salomon and H. Klar. Self-timed fully pipelined multipliers. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 45–55. Elsevier Science Publishers, 1993.
- [SMJ⁺94] Robert F. Sproull, Charles E. Molnar, Ian Jones, Bill Coates, and Jon Lexau. Counterflow pipeline processor project: Special invited session notes. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, November 1994.
- [SSL⁺92] Ellen M. Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Software Documentation Memorandum No. UCB/ERL M92/41, Electronics Research Laboratory, University of California, Berkeley., May 1992.
- [SSM94] Robert F. Sproull, Ivan E. Sutherland, and Charles E. Molnar. The counterflow pipeline processor architecture. *IEEE Design & Test of Computers*, 11(3):48–59, Fall 1994.
- [Sut89] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.
- [vB93] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.
- [vBB96] Kees van Berkel and Arjan Bink. Single-track handshaking signaling with application to micropipelines and handshake circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.
- [vBBK⁺94] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, and Frits Schalijs. A fully-asynchronous low-power error corrector for the DCC player. In *International Solid State Circuits Conference*, pages 88–89, February 1994.

- [vBBK⁺95] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, Frits Schlij, and Rik van de Wiel. A single-rail re-implementation of a DCC error detector using a generic standard-cell library. In *Asynchronous Design Methodologies*, pages 72–79. IEEE Computer Society Press, May 1995.
- [VBH⁺95] M. Valencia, M. J. Bellido, J. L. Huertas, A. J. Acosta, and S. Sanchez-Solano. Modular asynchronous arbiter insensitive to metastability. *IEEE Transactions on Computers*, 44(12):1456–1461, December 1995.
- [vBKR⁺91] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schlij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, pages 384–389, 1991.
- [vBR95] Kees van Berkel and Martin Rem. VLSI programming of asynchronous circuits for low power. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 152–210. Springer-Verlag, 1995.
- [vBS88] C. H. (Kees) van Berkel and Ronald W. J. J. Saeijs. Compilation of communicating processes into delay-insensitive circuits. In *Proc. International Conf. Computer Design (ICCD)*, pages 157–162. IEEE Computer Society Press, 1988.
- [Wal64] C.S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, EC-13:14–17, 1964.
- [WE93] Neil H. E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*. VLSI Systems Series. Addison-Wesley, 1993.
- [WFF94] Jyh-Ming Wang, Sung-Chuan Fang, and Wu-Shiung Feng. New efficient designs for XOR and XNOR functions on the transistor level. *IEEE Journal of Solid-State Circuits*, 29(7):780–786, July 1994.
- [WH91] Ted E. Williams and Mark A. Horowitz. A zero-overhead self-timed 160ns 54b CMOS divider. *IEEE Journal of Solid-State Circuits*, 26(11):1651–1661, November 1991.
- [WPS95] Ted Williams, Niteen Patkar, and Gene Shen. SPARC64: A 64-b 64-active-instruction out-of-order-execution MCM processor. *IEEE Journal of Solid-State Circuits*, 30(11):1215–1226, November 1995.
- [WS58] A. Weinberger and J.L. Smith. A logic for high-speed addition. In *National Bureau of Standards Circular 591*, pages 3–12, 1958.
- [YBA96] K. Y. Yun, P. A. Beerel, and J. Arceo. High-performance asynchronous pipeline circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.
- [YHJN95] J. T. Yantchev, C. G. Huang, M. B. Josephs, and I. M. Nedelchev. Low-latency asynchronous FIFO buffers. In *Asynchronous Design Methodologies*, pages 24–31. IEEE Computer Society Press, May 1995.

- [YS89] Jiren Yuan and Christer Svensson. High-speed CMOS circuit techniques. *IEEE Journal of Solid-State Circuits*, 24(1):62–70, February 1989.
- [YYN⁺90] Kazuo Yano, Toshiaki Yamanaka, Takashi Nishida, Masayashi Saito, Katsuhiko Shimohigashi, and Akihiro Shimizu. A 3.8ns CMOS 16x16b multiplier using complementary pass transistor logic. *IEEE Journal of Solid-State Circuits*, 25(2):388–395, February 1990.